Discussion Paper Series – CRC TR 224

Discussion Paper No. 522
Project C 01

Tranquilo: An Optimizer for the Method of Simulated Moments

Janoś Gabler[1]
Sebastian Gsell[2]
Tim Mensinger[3]
Mariam Petrosyan[4]

April 2024

[1]University of Bonn, Email: janos.gabler@gmail.com
[2]LMU Munich, Email: sebastian.gsell@econ.lmu.de
[3]University of Bonn, Email: tmensinger@uni-bonn.de
[4]University of Bonn, Email: mpetrosyan@uni-bonn.de

Support by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) through CRC TR 224 is gratefully acknowledged.

Tranquilo: An Optimizer for the Method of Simulated Moments*

Janoś Gabler[a], Sebastian Gsell[b], Tim Mensinger[a], Mariam Petrosyan[a]

[a] *University of Bonn*
[b] *LMU Munich*

March 29, 2024

We propose the tranquilo algorithm, a trust-region optimizer that aims to facilitate optimization problems that arise during the method of simulated moments estimation (MSM). The algorithm is particularly suited for this type of problem as it (1) can utilize the least-squares structure of the MSM problem, (2) can be parallelized on the level of the algorithm, and (3) can adaptively deal with noise in the objective function. The adaptive nature of tranquilo makes it particularly suited for domain experts such as statisticians and social science researchers without extensive training in numerical optimization. Extensive benchmarks show that tranquilo is competitive with state-of-the-art algorithms in noise-free settings and outperforms them in the presence of substantial noise.

1 Introduction

Economists frequently encounter "hard" optimization problems when fitting structural models to empirical data. By "hard" we mean that solving the optimization problem requires a significant amount of computation time, often hours or days; that manual intervention like tuning start values or adjusting algorithm parameters is required to obtain a solution; and that solving the optimization problems takes up a significant portion of the researcher's time. A prime example where such problems arise is the estimation of discrete choice models via the method of simulated moments (MSM).

Despite the prevalence of MSM estimation in structural papers (see Eisenhauer, Heckman, and Mosso (2015) for a review) and widely available anecdotal evidence that structural researchers

*Corresponding author: Janoś Gabler, janos.gabler@gmail.com. We are grateful for funding by Kenneth Judd and the Hoover Institution at Stanford University. Janoś Gabler is grateful for financial support from the German Research Foundation (DFG) through CRC-TR 224 (Project C01). Tim Mensinger thanks the Hausdorff Center for Mathematics for financial support. We thank Kenneth Judd, Michael Griebel, Hans-Martin von Gaudecker, and all other participants of the 2023 workshop "Numerical Methods and Applications in the Social Sciences" (University of Bonn) for helpful comments.

would love to spend less time on solving optimization problems, there are no specialized optimization algorithms that are tailored to the characteristics of MSM estimation problems.

The goal of our paper is to close this gap by proposing the *tranquilo* (**T**rust**R**egion **A**daptive **N**oise robust **QU**adrat**I**c or **L**inear approximation **O**ptimizer) algorithm – an optimizer that helps researchers solve hard optimization problems, as they arise during MSM estimation, faster and with less need for manual intervention. *tranquilo* is designed to take three main characteristics of MSM estimation problems into account:

First, MSM estimation problems are *nonlinear least-squares* problems. Least-squares optimization problems lie within the general class of *blackbox* optimization problems

$$\min_{l \leq x \leq u} f(x) \tag{DET}$$

where $f : \mathbb{R}^p \to \mathbb{R}$. In the least-squares case, the objective function f has the structure $f(x) = \sum_{j=1}^{k} r_j(x)^2$ and thus the optimization problem can be written as

$$\min_{l \leq x \leq u} \sum_{j=1}^{k} r_j(x)^2 \;=\; \min_{l \leq x \leq u} \|r(x)\|^2 \tag{DET-LS}$$

where $r(x) \equiv [r_1(x), \ldots, r_k(x)]^T : \mathbb{R}^p \to \mathbb{R}^k$ and $\|\cdot\|$ is the 2-norm of a vector. $r_j(x)$ is called a least-squares residual and $r(x)$ is called a residual vector.

It is easy to see that MSM problems are nonlinear least-squares problems. The objective function of an MSM problem is given by

$$f(x) \;=\; (m(x) - \hat{m})^T W (m(x) - \hat{m})$$

where x are the parameters to be estimated, $m(x)$ is a vector of simulated moments from the economic model, and $\hat{m}$ is a vector of empirical moments. W is a positive definite weighting matrix.

By defining $L = chol(W)$ as the lower triangular Cholesky factor of W, we can rewrite the objective function as

$$f(x) \;=\; (m(x) - \hat{m})^T L L^T (m(x) - \hat{m})$$

By defining $r(x) = L^T(m(x) - \hat{m})$, we can rewrite the objective function as

$$f(x) \;=\; r(x)^T r(x) \;=\; \sum_{j=1}^{k} r_j(x)^2 \;=\; \|r(x)\|^2$$

Which shows the least-squares structure of the MSM objective function.

There is a class of optimization algorithms that exploit the least-squares structure of the objective function, and it is a robust result that they outperform general-purpose algorithms when applicable (Cartis, Fiala, et al., 2019; Levenberg, 1944; Marquardt, 1963; Wild, 2017). A review of existing algorithms can be found in Section 2. *Tranquilo* builds on the class of derivative-free least-squares optimizers and extends them to meet the requirements of an efficient optimizer for MSM problems.

Second, MSM estimation problems as they arise in economics have an expensive objective function that is hard to parallelize.

In structural economic models, each evaluation of the MSM objective function first requires solving the model and then simulating data based on the solution. Solving a model can take anywhere from a few seconds to a few hours but is never in the range of milliseconds. Simulating data from a model is usually much faster but still adds some computation time.

An optimization algorithm that is designed for MSM estimation problems can thus assume that the evaluation of the objective function is a runtime bottleneck. Whenever there is a trade-off between reducing calculations done by the optimizer (e.g., doing linear algebra to calculate candidate points) vs. saving a few function evaluations, the optimizer should always prioritize saving function evaluations. This is in stark contrast to traditional objectives of optimization algorithms where often a significant amount of effort is spent on optimizing calculations done by the optimizer so that they perform well in cases where the evaluation of the objective function is very fast. .

Nowadays, most researchers have access to parallel hardware: 8 to 16 cores in a laptop; 16 to 64 cores in desktop computers and small servers, up to hundreds of cores in clusters. However, in many codebases, the objective function is not parallelized. This may partially be due to the fact that economists are not trained in parallel programming, but there are also inherent difficulties in parallelizing the solution of economic models. In any case, parallelizing the objective function would require a large time investment of the researcher.

This implies that an optimization algorithm should parallelize the evaluation of the objective function and thus shift the burden of parallel programming from researchers to algorithm developers. Algorithms that parallelize on the level of function evaluations exist (Lee and Wiswall, 2007), but to the best of our knowledge, none of them exploits the least-squares structure of the objective function.

A primary goal for *tranquilo* was to develop an algorithm that evaluates the objective function in batches and, instead of trying to minimize the number of function evaluations, tries to minimize the number of batches. The batch size corresponds to the number of available cores in the system. This design choice stems from the understanding that researchers typically do not benefit from idle cores on their computers. Instead, their priority is often to minimize the time it takes to solve the optimization problem.

Third, MSM estimation problems have a noisy objective function. Noise in the objective function means that we only observe noisy evaluations of the true objective function, but we are interested in finding the minimum or minimizer of the true objective function.

Thus, the problem to be solved is not given by equations DET or DET-LS but by their stochastic counterparts

$$\min_{l \leq x \leq u} \mathbb{E} f(x, \xi) \tag{STOCH}$$

$$\min_{l \leq x \leq u} \mathbb{E} \|r(x, \xi)\|^2 = \min_{l \leq x \leq u} \mathbb{E} \sum_{j=1}^{k} r_j(x, \xi_j)^2 \tag{STOCH-LS}$$

In the method of simulated moments, the noise in the objective function comes from the fact that we are simulating data of a stochastic model. As researchers, we can influence the amount of noise in the objective function by increasing the number of simulation draws. However, this comes at the cost of increased computation time; in many cases, reducing the amount of noise to a level that is acceptable for standard optimizers is usually prohibitively expensive. This is especially the case in dynamic discrete models where initially small random influences may propagate over time and thus lead to large differences in the simulated data of later periods.

A common approach to deal with noisy objective functions inolves fixing the seed of the random number generator and using the same random draws in each iteration of the optimizer. In general, this approach is not suitable for solving STOCH or STOCH-LS. The reason is that the optimizer will be influenced by lucky draws under the chosen seed and has no chance to optimize the true objective function, i.e., the expected value of the observed function $f(x, \xi)$.

Moreover, in dynamic discrete models, this approach of fixing the seed also fails to produce a well-behaved objective function. While it renders the objective function deterministic, it can introduce discontinuities and local optima even if the underlying true objective function is smooth.

In order to partial out noise, any optimizer that aims to solve STOCH or STOCH-LS has to evaluate the objective function more often than an equivalent optimizer for deterministic problems. Typically, such optimizers ask a user to specify the number of function evaluations, either as a fixed sequence or a function that depends on the iteration counter and other internal variables of the optimizer. This does not only require knowledge about the inner workings of the optimizer but is very hard to do in practice, as the ideal number of function evaluations depends on problem properties that are not known ex-ante. We illustrate this in Subsection 5.1. Usually, the ideal sequence is increasing in the iteration counter. Choosing too many evaluations slows down the progress. Choosing too few can lead to a catastrophic failure of the optimizer.

A primary goal of our optimizer is, therefore, to determine the optimal number of function evaluations in an adaptive fashion without requiring any user-provided information on the amount or type of noise in the objective function.

While *tranquilo* is tailored to MSM estimation problems as they arise in economics, it is not limited to these. Problems with the same characteristics are also encountered in other fields. Prime examples are design optimization in engineering or calibrating epidemiological models to empirical data. In fact, one of the main motivations for developing *tranquilo* comes from an epidemiological model (Gabler, Raabe, et al., 2022).

To summarize the contributions on a technical level, some familiarity with derivative-free trust-region optimizers is required. We describe the basic intuition of derivative-free trust-region optimization in Section 2 and refer the reader to Conn, Gould, and Toint (2000) for a more detailed introduction. The technical contributions are as follows.

First, We take a fairly standard trust-region framework for nonlinear least-squares optimizers (see for example, Conn, Gould, and Toint (2000)) and reformulate it in a modular fashion that allows us to replace individual components of the algorithm in order to customize it to the characteristics of MSM estimation problems. Besides the obvious benefit of easing the implementation, this modularization generates important insights. For example, we show that the fundamental difference between a scalar and least-squares version of *tranquilo* is concentrated in one single step (see Section 15), which is not obvious when looking at other codebases that implement scalar and least-squares versions of an algorithm (e.g., Cartis, Fiala, et al. (2019) implement the scalar *PY-BOBYQA* and least-squares *DFO-LS* in two separate codebases even though they highlight the similarity of both algorithms in their paper).

Second, we add parallelization capabilities to the trust-region framework. Some parts of derivative-free trust-region algorithms, such as the evaluation of the objective function on an initial set of points, are embarrassingly parallel and have been parallelized in other algorithms (e.g., Cartis, Fiala, et al. (2019)). We add two new ideas for more efficient parallelization: The first is a parallel line search that tries out multiple step lengths in the search direction obtained by solving the trust-region subproblem. The second is speculative sampling: While doing the function evaluation(s) needed to decide whether a candidate point is accepted, we already sample points that would be helpful in the next iteration if the candidate point is accepted, and evaluate the objective function on those points. Both strategies have diminishing returns if many cores are available. Therefore, we find that a combination of both approaches works best.

Third, we propose novel ways of adaptively determining how many function evaluations are needed to average out the noise just enough so that the optimizer can make progress. We distinguish two different situations within each iteration.

In the *model building phase*, we need to determine how often the objective function should be evaluated on each model point. The goal here is to build a model that is as cheap as possible but good enough to send us in the right direction. We treat the error that derives from noise in a similar

form as the error that derives from approximating a general nonlinear function over the trust-region with a low-order polynomial. To this end, we introduce a new measure of model quality ρ^{noise} that measures how strongly random error impedes the surrogate model's ability to produce good candidate points. This measure is then used to adjust the number of repeated function evaluations at each model point. In this sense, it is similar to the traditional measure of model quality ρ that is used to adjust the trust-region radius. The calculation of ρ^{noise} is based on a simulation approach that is computationally costly compared to an iteration of a normal trust-region algorithm but small compared to a single evaluation of the objective function in typical applications.

In the *acceptance phase*, we need to determine how many function evaluations are needed to decide whether the candidate point is actually an improvement over the currently accepted point. We use power analysis to determine the minimal number of additional function evaluations on both the candidate point and the currently accepted point that are needed to decide which point is better given some pre-specified statistical certainty. The approach takes the existing number of evaluations on both points as well as the expected improvement – a by-product of the trust-region step – into account.

Both approaches require an estimate of the variance of the noise in the objective function. We estimate this variance from existing function evaluations on points in a neighborhood of the current trust-region. By doing multiple function evaluations on the start parameters, we can guarantee that a sufficient number of function evaluations is available in all iterations and no extra function evaluations are needed for the noise estimation. This approach treats the noise variance as locally constant over the trust-region but otherwise accommodates both additive and multiplicative noise as well as mixtures thereof and does not require the user to specify which type of noise is present.

Fourth, we make *tranquilo* (Gabler, Gsell, et al., 2024) available as an open-source Python package that can be used in isolation or via the estimagic package (Gabler, 2022).

Tranquilo builds heavily on previous algorithms and the literature on derivative-free optimization (Cartis, Fiala, et al., 2019; Conn, Gould, and Toint, 2000; Powell, 2009; Wild, 2017). While this literature uses terminology that is not commonly familiar to economists, it is surprising that large parts of *tranquilo* can be understood in terms of concepts that are familiar to economists: Power analysis is routinely used to determine sample sizes in empirical work; in *tranquilo*, it is used to determine the number of function evaluations for accepting a candidate point. Using model-based simulations to determine optimal policies is the core business of structural economists; in *tranquilo* we use them to determine an optimal policy for the number of function evaluations used in the model building phase. Finally, ordinary least-squares regression is the workhorse method for every empirical economists; in *tranquilo* we use it to fit linear or quadratic approximations to a general nonlinear function.

We benchmark *tranquilo* against existing solvers on the Moré-Wild benchmark set Moré and Wild (2009), which is the standard benchmark set for derivative-free least-squares solvers. To assess the

performance of *tranquilo* on noisy problems, we add artificial noise to the objective functions of the benchmark set.

In a baseline setting without noise and parallelism, the least-squares and scalar versions of *tranquilo* are competitive with comparable existing solvers. The least-squares version is slightly slower than the best existing least-squares solver *DFO-LS*, but faster than *POUNDERS*, which came out as the best optimizer in Eisenhauer, Heckman, and Mosso (2015). The scalar version is slightly slower than the NlOpt implementation of *BOBYQA* but beats the scipy and NlOpt implementations of Nelder-Mead as well as the NAG implementation of *BOBYQA*. While this is not the primary use case for *tranquilo*, it is reassuring that *tranquilo* is competitive with existing solvers in the baseline setting. The full results and details of the benchmarking procedure for this setting can be found in Subsection 3.3.

To assess parallel performance, we compare *tranquilo* versions with 1, 2, 4, and 8 cores against each other. We also include *DFO-LS* as a reference. Importantly, this time, the goal is not to minimize the number of function evaluations but the number of batches, where each batch is a set of function evaluations that can be run in parallel. As before, we find that *DFO-LS* is faster than the serial version of *tranquilo*, but with two cores, *tranquilo* is already considerably faster than *DFO-LS*. Adding more cores keeps improving the performance of *tranquilo*, and the 8-core version is the fastest solver for more than 80% of the problems in the benchmark set. The full results and details of the benchmarking procedure for this setting can be found in Subsection 4.2.

In a noisy setting, we compare *tranquilo* against *DFO-LS* – the only other derivative-free least-squares solver that is designed to handle noisy objective functions. Since *DFO-LS* requires the user to specify the number of function evaluations at each parameter vector, we compare *tranquilo* against multiple variants of *DFO-LS*. We restrict our attention to a fixed number of function evaluations because correctly guessing sequences that vary in each iteration is very hard to do in practice. Compared to the noisy benchmarks in Cartis, Fiala, et al. (2019), we use a much larger amount of noise.

We find that *tranquilo* outperforms all configurations of *DFO-LS* in the noisy setting. While *DFO-LS* configurations with few function evaluations per parameter vector solve some problems very quickly, they fail to solve others. Configurations with many function evaluations per parameter vector solve more problems but are very slow. Due to its adaptive nature, *tranquilo* is able to solve problems quickly while still being robust to solve many problems. The full results and details of the benchmarking procedure for this setting can be found in Subsection 5.4.

The remainder of the paper is structured as follows: Section 2 reviews core concepts and terminology of derivative-free optimization and discusses how existing algorithms relate to *tranquilo*. Section 3 describes the modular formulation of our general trust-region framework and discusses the implementation of each component for the baseline case without noise and parallelization. It also shows the results of benchmarking *tranquilo* against existing solvers in this setting. Section 4 explains our two ideas for improving the parallelization of derivative-free trust-region optimizers

and shows the speed-up we achieve via parallelization. Section 5 describes our approaches for noise handling as well as the corresponding benchmarks. Section 6 concludes.

2 Literature review

The literature review is split into two parts. The first reviews important concepts of derivative-free optimization and is dedicated to readers with little or no background in optimization. We introduce all essential concepts needed to understand the rest of the paper, as well as the technical description of contributions in the introduction. The second part reviews related algorithms and identifies gaps in the literature that are filled by *tranquilo*.

2.1 Concepts of derivative-free optimization

Local and global optimization. In economics and statistics, it is often the goal to find a global minimum of a scalar objective function defined on $\mathbb{R}^p$. Without further assumptions, this is an impossible task, as the only way to guarantee that a global optimum was found is to evaluate the objective function at all points in $\mathbb{R}^p$.

There are two ways to solve global optimization problems in practice: Global optimizers or local optimizers in a multistart framework.

Global optimizers require finite bounds for all parameters and sample the parameter space. The simplest algorithms are random search and grid search; other algorithms sample candidate points in more sophisticated ways. Global algorithms often yield relatively imprecise solutions that must be refined with a local optimizer. Moreover, they suffer from the curse of dimensionality, i.e., they become extremely expensive as soon as there are more than a handful of parameters. A big drawback of global optimizers is that they typically do not exploit any known properties of the objective function. For example, we are not aware of global optimizers that exploit the least-squares structure. Without further precautions, global optimizers are also not robust to noise in the objective function. A simple example of this is random search. While random search is a very robust global optimizer for deterministic functions, it breaks down if there is considerable noise in the objective function, as it might select a point that just had a lucky draw.

Multistart frameworks run local optimizers from multiple starting points. While any single optimization run might get stuck in a local minimum, the hope is that the best local minimum is also the global minimum. As with global optimizers, multistart frameworks come without guarantees that the global minimum was found. Their biggest advantage is that they work with any local optimizers and, thus, can exploit known properties of the objective function, such as the least-squares structure. As long as the local optimizer is robust to noise, multistart frameworks are also robust to noise.

Given these trade-offs, we decided to develop a local optimizer. If a global optimum is required, we recommend to run *tranquilo* in an efficient multistart framework such as tiktak (Arnoud, Guvenen, and Kleineberg, 2019).

Derivative free optimization. Local optimizers move iteratively through the parameter space of an optimization problem to find a parameter vector that minimizes the objective function. Thus, in each iteration, the algorithm needs to decide on a search direction and a step size. In derivative-based optimization, the search direction is usually based on the gradient of the objective function, and the step size is chosen based on its Hessian (see Nocedal and Wright (2006) for examples).

While this approach is very successful, it requires a means to evaluate the gradient and, potentially, the Hessian of the objective function. Whenever one has access to the objective function itself, a way to get at its derivatives is to use finite differences. However, this approach is very expensive. If there are p parameters, calculating a gradient via finite differences takes at least p additional evaluations of the objective function, and second derivatives are even more expensive.

Gradient-free optimizers do not make direct use of the derivatives of the objective function. By not using derivatives, their goal is to be faster than a gradient-based optimizer employing finite differences. There are different classes of gradient-free optimizers. Each of them uses a different approach to finding a search direction and a step size without using the derivatives of the objective function. We restrict our attention to the class of derivative-free trust region optimizers. For an overview of other approaches, see Larson, Menickelly, and Wild (2019).

Importantly, many derivative-free optimizers assume that the derivatives of the objective function exist. They simply do not use them because they are too expensive to evaluate. The existence of derivatives is needed for convergence proofs. In practice, some derivative-free optimizers work even if the derivatives do not exist.

The basic idea of trust-region optimization. An important class of derivative-free optimizers are trust-region methods (Conn, Gould, and Toint, 2000; Nocedal and Wright, 2006). One iteration of a prototypical trust-region algorithm looks as follows

(1) Given a current parameter vector x_t as trust-region center and a radius Δ_t, form a surrogate model M_t that approximates the objective function inside the trust-region. The surrogate model is usually a quadratic model or some other low-order polynomial.

(2) Find the minimizer of the surrogate model using a specialized optimizer that is tailored to the functional form of the surrogate model. This minimizer becomes a candidate step.

(3) Evaluate the objective function at the candidate point and accept or reject the candidate point.

(4) Adjust the trust-region radius for the next iteration based on a measure of progress.

The basic idea of a trust-region optimizer is to iteratively replace an expensive objective function that is hard to optimize with a local surrogate model that can be optimized very cheaply. The acceptance decision and trust-region management play an important role in ensuring that the model approximates the function well enough.

Surrogate models. Different trust-region optimizers form the surrogate models in different ways. Derivative-based methods use the gradient and Hessian of the objective function at x_t to form a second-order Taylor expansion that serves as the surrogate model. To save costly evaluations of the Hessian, some optimizers use approximations to the Hessian. A robust result in the literature is that (underdetermined) quadratic surrogate models work best (Conn, Gould, and Toint, 2000). Linear surrogate models have no internal minimum and can thus only suggest candidate points on the boundary of the trust-region, which makes them unsuitable for choosing good step lengths. Higher-order polynomials are not just too expensive to form but also too hard to optimize.

Derivative-free optimizers form surrogate models by evaluating the objective function on a sample of points and forming a quadratic model by interpolation or regression. The points are chosen carefully based on geometric considerations to maximize the model's approximation accuracy. To save function evaluations, only a few points in the sample are replaced in each iteration. Fully determined quadratic interpolation models require the function to be evaluated at $\frac{(p+1)(p+2)}{2}$ points. Since this number grows quickly in p, many algorithms use underdetermined interpolation models based on $2p+1$ function evaluations. The remaining degrees of freedom are resolved by choosing a solution to the interpolation conditions that minimize the Frobenius norm of the model Hessian or the Frobenius norm of the change in the model Hessian between iterations. This idea was first popularized by Powell in the *NEWUOA* and *BOBYQA* algorithms (Powell, 2009; Powell, 2006) and has since been used by many others (see Larson, Menickelly, and Wild (2019) for a review).

A special case are derivative-free trust-region methods for least-squares problems. Instead of forming just one surrogate model for the function value, they form a surrogate model for each least-squares residual. The surrogate models for the residuals are then aggregated into a surrogate model for the actual function value. While there are no proofs that this approach works better than forming scalar surrogate models directly, a vast amount of benchmarks shows that as few as $p+1$ function evaluations can be enough to create useful surrogate models using this principle (see for example Cartis, Fiala, et al. (2019) and Cartis and Roberts (2019)).

Trustregion radius management. The surrogate models in trust-region optimizers only approximate the objective function locally. Using simple models like quadratic ones can, therefore, be justified by Taylor's theorem. This shows that the trust-region radius plays a central role in governing the approximation quality. If the radius is large, the optimizer can make large steps, but the model might be a poor approximation to the objective function. Making the radius smaller increases the model accuracy at the cost of slower progress.

It is very important that the model only has to be good enough to make progress, and it is not an explicit goal to minimize the overall approximation error on the trust-region. The radius adjustment is, therefore, based on a measure of model quality that specifically takes into account how well the model predicts good candidate points

$$\rho_t \equiv \frac{f(x_t^*) - f(x_t^* + s_t)}{M_t^s(x_t^*) - M_t^s(x_t^* + s_t)} = \frac{\text{Actual Improvement}}{\text{Expected Improvement}} \tag{2.1}$$

The basic idea is then as follows: If ρ is large, the model worked well in predicting a descent direction, and the radius can be increased or kept constant. If ρ is small, the radius has to be reduced in order to improve the model quality in the next iteration. If suitable surrogate models are used and regularity conditions are fulfilled, Taylor-like error bounds guarantee that a good approximation quality can be achieved by making the radius small enough. The actual radius adjustment is slightly more complex and depends on additional quantities and conditions. Several methods are discussed in Conn, Gould, and Toint (2000).

Convergence. The word *convergence* is used for two very different things: In the theoretical literature, a convergence proof means that a mathematical algorithm is guaranteed to find a local optimum or stationary point if run for long enough. Among practitioners, convergence means that an algorithm stopped the optimization process because a condition was achieved. Since those conditions can usually be set by a user, reaching them is not a strong guarantee that an optimum has been found, and practitioners should always verify that convergence was not spuriously induced by weak convergence criteria.

Tranquilo is loosely based on a trust-region framework for which a convergence proof exists (Conn, Gould, and Toint, 2000), and the components that play a central role in the convergence proof (e.g., solvers for the surrogate problem and trust-region radius handling) are fairly standard. However, *tranquilo* is meant as an algorithm for practitioners, and we do not make an attempt at extending the convergence proof to cover the modifications we propose in *tranquilo*. Instead, we rely on extensive benchmarks to show the practical performance of *tranquilo*.

2.2 Related algorithms

We restrict our attention to derivative-free trust-region methods for bound-constrained optimization. A more comprehensive overview discussing other methods can be found in Larson, Menickelly, and Wild (2019).

While derivative-free trust-region optimizers based on quadratic models have been used since the early 1970s (Winfield, 1973), the interest in these methods has been revitalized by the influential work of Powell. An important contribution of Powell was the introduction of underdetermined interpolation for the construction of quadratic surrogate models –first introduced in the *NEWUOA* and *BOBYQA* algorithms– which drastically improve the efficiency for higher dimensional problems (Powell, 2009; Powell, 2006). As an algorithm that supports bound constraints, *BOBYQA* can be seen as the direct predecessor of most algorithms that we discuss in this section.

The *BOBYQA* algorithm (Powell, 2009) maintains a sample of $2p+1$ points that are used to form a quadratic surrogate model. The model is fit using underdetermined interpolation. The remaining degrees of freedom are resolved by choosing the solution to the interpolation conditions that minimize the Frobenius norm of the change in the model Hessian between two iterations. Between two iterations, at most one model point is replaced. The replacement point is chosen to maximize the stability of the model. Several variants of the *BOBYQA* algorithm are available as open-source software and compare very favorably against other derivative-free optimizers like the Nelder-Mead algorithm (see Subsection 3.3).

The *DFBOLS* (Zhang, Conn, and Scheinberg, 2010) and *POUNDERS* algorithm (Wild, 2017) can be seen as a translation of *BOBYQA* to least-squares problems. Both algorithms use $2p+1$ interpolation points and the same underdetermined interpolation method as *BOBYQA*. The main difference is that they construct one quadratic surrogate model for each least-squares residual and aggregate those models into a quadratic model for the objective function. The aggregation method differs between the optimizers: *POUNDERS*' aggregation method can be described as a Full-Newton approach whereas *DFBOLS* incorporates elements from a Gauss-Newton approach. The *POUNDERS* algorithm is available as a pure Python implementation in the estimagic library. A C implementation of *POUNDERS* is available in the toolkit for advanced optimization (TAO) (Dener et al., 2021). *DFBOLS* is available as Fortran code. The performance of *DFBOLS* and *POUNDERS* is expected to be very similar (Wild, 2017). Due to the lack of a *DFBOLS* implementation with Python bindings, we only compare *tranquilo* to *POUNDERS*.

DFO-LS (Cartis, Fiala, et al., 2019) is another derivative-free trust-region method for least-squares problems. The key difference is that *DFO-LS* uses only $p+1$ interpolation points and fits fully determined linear surrogate models for each residual. Those linear models are then aggregated into a quadratic model for the objective function. This change drastically improves *DFO-LS*'s performance for larger problems. We use the same approach in *tranquilo*. On top of this change, *DFO-LS* introduces several new features: First, a fast start option tries to make progress before there are enough function evaluations to fit an initial model. Second, there is a heuristic that detects whether the trust-region radius collapsed due to the presence of noise and if so, the algorithm is automatically restarted. Third, the user can specify sequences that control how often a noisy objective function should be evaluated. The sequence can depend on several quantities, among them the iteration counter and a restart counter. The same new features are also available in *Py-BOBYQA*, which is developed in the same paper and works for scalar objective functions. The performance of *DFO-LS* is excellent (see Subsection 3.3) and we use it as the main benchmark for *tranquilo*. *Py-BOBYQA* is also included in the benchmarks but performs slightly worse than other *BOBYQA* implementations. Both algorithms are available as standalone Python packages.

The main problem of derivative-free trust-region optimizers applied to noisy objective functions is that the trust-region radius collapses to zero. This is caused by bad candidate points from noise-affected surrogate models and spurious rejections due to unlucky draws in the acceptance evaluation. While *DFO-LS* is the only least-squares optimizers for noisy objective functions that we are

aware of, there are several optimizers for scalar objective functions that employ strategies to avoid the collapsing of the radius.

SNOWPACK (Augustin and Marzouk, 2017) ties the trust-region radius management to an estimate of the noise in function evaluations. Moreover, it uses Gaussian process models instead of quadratic interpolation models to reduce the effect of noise.

Shashaani, Hashemi, and Pasupathy (2018) recognize that user-specified sequences for the number of function evaluations needed to average out noise are impractical and propose the *ASTRO-DF* algorithm that uses adaptive sampling: The number of evaluations is increased until an estimated standard error falls under a threshold. The threshold is a fixed factor of the squared trust-region radius. This incorporates the idea that smaller trust-region radii require more precise models. Moreover, it prevents the radius from shrinking too much before a good model quality has been achieved. The adaptive sampling in *ASTRO-DF* is, however, not based on the magnitude of the function evaluations. *ASTRO-DF* is available as part of the simopt library, where it can be benchmarked against other simopt optimizers. We currently exclude *ASTRO-DF* from our benchmarks because we could not get it to solve our benchmark problems precisely enough, but we want to exclude all errors that might be caused on our side before drawing any conclusions.

Parallelization on the algorithm level has not been a focus of the literature on derivative-free trust-region optimizers or derivative-free least-squares optimizers. However, there are parallel direct search algorithms for scalar problems.

Lee and Wiswall (2007) introduce a parallel version of the *Nelder-Mead* simplex algorithm. The classical *Nelder-Mead* algorithm maintains a set of $p+1$ points that are used to form a simplex in the parameter space. In each iteration, the worst point is replaced by a new point. There are different strategies for calculating the new point, which are selected based on the function values. The parallel version replaces more than one point in each iteration and evaluates the objective functions on all new points in parallel. Depending on the function value, an initial candidate for a new point might be rejected, and a second function evaluation is necessary before a new point is accepted. The empirical results in Lee and Wiswall (2007) show strong gains in efficiency, which are sometimes substantially larger than the number of processors. They explain this by the fact that the parallel version might sometimes create better search directions than the serial one. An implementation of the parallel *Nelder-Mead* algorithm is available in the estimagic library. We are currently working on incorporating it into our benchmarks.

3 Tranquilo core algorithm

In this section, we describe a core version of the *tranquilo* algorithm that is suitable for solving the deterministic nonlinear least-squares problem DET-LS as well as the deterministic scalar problem DET without using parallelization. The extension to the parallel case is described in Section 4. The extension to the stochastic case is described in Section 5.

The structure is as follows: In Subsection 3.1, we describe our modular formulation of a general trust-region algorithm that formalizes the interface of components in the algorithm. Most components are mathematical functions that have a one-to-one correspondence in the Python implementation of the algorithm. At this stage, we only describe the inputs and outputs of functions and are agnostic about their inner workings. In Subsection 3.2, we change our focus and describe the algorithmic implementation of each component. We focus on the deterministic and serial case, and draw ample comparisons to existing algorithms. In Subsection 3.3, we describe how we benchmark optimizers and show how *tranquilo* compares to other algorithms.

3.1 The trust region framework

In this section, we lay out the general trust-region framework of the *tranquilo* algorithm in a modular fashion with a high level of abstraction. Doing so allows us to describe the concrete implementation of our baseline algorithm as well as its extensions to the parallel and noisy case clearly and without repeating what stays unchanged. The full algorithm is described in Algorithm 1. A lookup table for our notation can be found in Appendix A.

Tranquilo is flexible because it is made up of *replaceable components*. By replaceable component, we mean a function that takes a specified set of inputs and produces a specified set of outputs. A simple example of a replaceable component is a *Sampler*, which takes existing points, a trust-region, and a target sample size as inputs and produces a set of new points as output. How the new points are created is not specified and varies across different samplers. Importantly, all other parts of *tranquilo* will work with any sampler that conforms to the specified set of inputs and outputs. This has, of course, a clear mapping to the Python implementation of *tranquilo*: For every replaceable component, we implement several different versions that a user of the algorithm can select by providing the name of that version. Advanced users can go beyond what we offer and implement their own versions of components.

A full list of replaceable components and a definition of their interfaces can be found in Table A.4. The implemented versions of each component are described in Sections 3.2, 4.1, and 5. Before looking at these implementations, we first describe how the different components interact to create the *tranquilo* algorithm.

Algorithm 1: *Tranquilo* algorithm

Input: Starting point x_0^*, initial trust-region radius Δ_0^{region}, target sample size n^{target}, search factor γ^{search}, minimum step size s^{min}, sample increment n_{stag}^{drop}, maximum number of iterations t^{max}, maximum number of trials to avoid stagnation n_{stag}^{max}, lower and upper bounds l and u.

Initialize history with $\mathscr{H}_0 = \{(x_0^*, r(x_0^*))\}$
Initialize vector model M_0^v with intercept terms at $r(x_0^*)$ and all other coefficients set to zero
for t=0,1,...,t^{max} **do**
    Calculate the search radius $\Delta_t^{search} = \gamma^{search} \Delta_t^{region}$
    Calculate the effective trust-region R_t based on x_t^*, Δ_t^{region}, l and u
    Scan the history for existing points $\mathscr{X}_t^{existing} = \{x \in \mathscr{H}_t : \|x_t^* - x\| \leq \Delta_t^{search}\}$
    Filter existing points: $\mathscr{X}_t^{filtered} = Filter(\mathscr{X}_t^{existing})$
    if $|\mathscr{X}_t^{filtered}| < n^{target}$ **then**
        Sample $n^{target} - |\mathscr{X}_t^{filtered}|$ new points in the trust-region: $\mathscr{X}_t^{new} = Sample(\mathscr{X}_t^{filtered}, R_t, n^{target})$
        $\mathscr{X}_t^{model} = \mathscr{X}_t^{filtered} \cup \mathscr{X}_t^{new}$
    else
        $\mathscr{X}_t^{model} = \mathscr{X}_t^{filtered}$
    end
    Build a vector model $M_t^v = Fit(\mathscr{X}_t^{model}, \mathscr{R}_t^{model}, M_{t-1}^v, R_t)$
    Aggregate the vector model: $M_t^s = Aggregate(M_t^v)$
    Solve the surrogate problem: $s_t = Subsolve(M_t^s, R_t)$
    while $|\mathscr{X}_t^{model}| > n^{target}$ *and* $\|s_t\| \leq s^{min}$ **do**
        Reduce the sample: $\mathscr{X}_t^{reduced} = Drop(\mathscr{X}_t^{model}, n_{stage}^{drop}, \Delta_t^{region})$ and set $\mathscr{X}_t^{model} = \mathscr{X}_t^{reduced}$
        Build a vector model $M_t^v = Fit(\mathscr{X}_t^{model}, \mathscr{R}_t^{model}, M_{t-1}^v, R_t)$
        Aggregate the vector model: $M_t^s = Aggregate(M_t^v)$
        Solve the surrogate problem: $s_t = Subsolve(M_t^s, R_t)$
    end
    $n_{stag} = 0$
    while $\|s_t\| \leq s^{min}$ *and* $n_{stag} \leq n_{stag}^{max}$ **do**
        Reduce the sample: $\mathscr{X}_t^{reduced} = Drop(\mathscr{X}_t^{model}, n_{stag}^{drop}, \Delta_t^{region})$
        Sample new points in the trust-region: $\mathscr{X}_t^{new} = Sample(\mathscr{X}_t^{reduced}, R_t, n^{target})$ and set $\mathscr{X}_t^{model} = \mathscr{X}_t^{reduced} \cup \mathscr{X}_t^{new}$
        Build a vector model $M_t^v = Fit(\mathscr{X}_t^{model}, \mathscr{R}_t^{model}, M_{t-1}^v, R_t)$
        Aggregate the vector model: $M_t^s = Aggregate(M_t^v)$
        Solve the surrogate problem: $s_t = Subsolve(M_t^s, R_t)$
        $n_{stag} = n_{stag} + 1$
    end
    Calculate $\Delta M_t^s = M_t^s(x_t^*) - M_t^s(x_t^* + s_t)$
    Accept or reject the step and calculate a measure of progress $(x_{t+1}^*, \rho_t) = Accept(x_t^*, s_t, \Delta M_t^s)$
    Adjust the trust-region radius: $\Delta_{t+1}^{region} = AdjustRadius(\Delta_t^{region}, \rho_t, s_t)$
    if $x_{t+1}^* \neq x_t^*$ *and* $Converged(\mathscr{H}_t, M_t^s, x_t^*, x_{t+1}^*)$ **then**
        break
    end
end

At the beginning of *tranquilo*, we are equipped with a starting point $x_0^* \in \mathbb{R}^p$, an initial radius $\Delta_0^{region} > 0$, as well as the lower and upper bounds of the optimization problem $l, u \in \mathbb{R}^p$. Together, those quantities define the initial trust-region. Moreover, we have several algorithm constants like the target sample size n^{target}, the search factor γ^{search}, the minimum step size s^{min}, the sample increment n_{stag}^{drop}, the maximum number of iterations t^{max}, and the maximum number of trials to avoid stagnation n_{stag}^{max}. For now, we abstract from constants that are only used by the specific implementation of components.

The algorithm starts by evaluating the objective function at the starting point and initializing the history of function evaluations with $\mathscr{H}_0 = \{(x_0^*, r(x_0^*))\}$ —If it is clear from the context, we sometimes write $x \in \mathscr{H}$ instead of $(x, r(x)) \in \mathscr{H}$. The history of function evaluations is scanned at the beginning of each iteration to find points on which the objective function has previously been evaluated and which are inside or near the current trust-region. Moreover, we initialize a surrogate model for the least-squares residuals M_0^v to equal the constant $r(x_0^*)$ for all points inside the trust-region. We call this a vector model to distinguish it from the aggregated scalar model that approximates the objective function instead of the residuals.

Before the first trust-region iteration, we calculate the effective trust-region R_t which is the subset of the parameter space in which new points can be sampled and to which the solution of the trust-region subproblem will be constrained. If no bounds are binding, the effective trust-region is just the trust-region, i.e., a ball with center x_t^* and radius Δ_t^{region} in Euclidean norm. If bounds are binding, we switch to a hypercube trust-region with the same volume as a ball of radius Δ_t^{region}. The hypercube is also centered at x_t^* and clipped to comply with the bounds of the optimization problem. Note that a hypercube can be viewed as a ball under the maximum-norm. To avoid confusion, we stick to saying ball for spherical regions and hypercube for cubical regions. Other trust-region algorithms that allow for bound constraints (e.g., Wild (2017)) work with a radius in maximum-norm from the beginning. However, we found that switching between the two shapes yields a better performance in benchmarks.

At the beginning of each iteration, we scan the history of function evaluations for points that lie within the search radius of the current trust-region center. These points can be re-used when building the surrogate model. Next, the set of points is filtered. The filtering step is the first replaceable component of the *tranquilo* algorithm. Having a filtering step is a design choice inspired by the following seemingly counter-intuitive observation: A sample size that is *too large* can actually make surrogate models worse (Larson, Menickelly, and Wild, 2019; Powell, 2009). The filtering step provides the option to discard points that are too close to each other or too close to the trust-region center. In our practical experiments, however, we could not confirm this observation and use the identity function as a filter. Other filters we tried and implemented are described in Subsection 3.2.1.

The scanning and filtering approach differs from other trust-region algorithms that do not maintain a full history and only store a fixed-size set of model points (Cartis, Fiala, et al., 2019; Powell, 2009; Wild, 2017). To add a new point, old ones have to be discarded. We find the scanning and filtering approach appealing because it allows the user to warm-start the algorithm with a database of previous function evaluations and for costly objective functions, the memory overhead of storing the full history is not a concern.

If the number of filtered points is smaller than the target sample size n^{target}, we sample new points in the current trust-region until we reach the target sample size. The sampling step is another replaceable component of the *tranquilo* algorithm. The sampling can be based on the geometry of the existing points, and all sampled points must lie inside the effective trust-region. We discuss the sampling strategies we implemented in Subsection 3.2.2.

Scanning, filtering, and sampling leave us with a set of model points $\mathcal{X}_t^{model}$. After evaluating the objective function on the newly sampled points, we can also construct a corresponding set of least-squares residuals $\mathcal{R}_t^{model}$. These can be used to fit a vector model M_t^v. Fitting is another replaceable component of the *tranquilo* algorithm that allows us to nest the fitting strategies of different algorithms in a simple way. In addition to $\mathcal{X}_t^{model}$ and $\mathcal{R}_t^{model}$, the fitting method needs two more ingredients: First, the previous vector model M_{t-1}^v, which can, for example, be used to penalize changes in the model Hessian and second, the effective trust-region, which is used to scale the model to a unit-ball or unit-hypercube –depending on the shape of the trust-region– for numerical stability.

Fitting methods can differ by the type of vector model they fit (e.g., linear or quadratic), by the way they resolve degrees of freedom in the case of underdetermined interpolation (e.g., penalize Hessian terms or changes in Hessian terms), and by the way they do the actual fitting (e.g., ordinary least squares, least absolute deviation, lasso or ridge regression). To ensure that the model fitting is well-defined, all fitting methods must work for underdetermined, just-determined, and over-determined fitting problems. We discuss the fitting methods we implemented in Subsection 3.2.3.

The next step is to aggregate the vector model M_t^v (the surrogate model that approximates the residual function $r(x)$) into a scalar model M_t^s (the surrogate model that approximates the objective function $f(x)$). The minimizer of the scalar surrogate model will become the next candidate point. The aggregation step is another replaceable component of the *tranquilo* algorithm, but it is important that the fitting step, which decides whether linear or quadratic residual models are built, and the aggregation step are compatible and produce a well-defined quadratic scalar model.

By choosing appropriate pairs of fitting and aggregation methods, we can nest the fitting and aggregation strategies of different algorithms; such as fitting linear residual models and aggregating them into a quadratic scalar model (Cartis, Fiala, et al., 2019) or fitting quadratic residual models and aggregating them into a scalar model (Wild, 2017; Zhang, Conn, and Scheinberg, 2010).

By treating scalar optimization problems as outputting a vector of size one and using an identity function as the aggregation method, we can even nest the fitting approach of scalar algorithms like *BOBYQA* (Powell, 2009). Even though our primary focus is developing a least-squares optimizer, we show that the resulting algorithm is competitive with other derivative-free scalar optimizers (see Subsection 3.3). Another possible extension would be a dedicated optimizer for likelihood functions that leverages the information matrix equality to construct a quadratic scalar model from linear surrogate models. This would be a derivative-free analog of the popular *BHHH* algorithm (Berndt et al., 1974).

Using the scalar model M_t^s, we solve the trust-region subproblem to obtain a candidate step length s_t. The subsolver is again a replaceable component of the *tranquilo* algorithm. After extensive ex-

perimentation, we found that two common methods work best: If bounds are binding, we use the *BNTR* algorithm, otherwise we use the *GGTPAR* algorithm. Both solvers are also available in the *POUNDERS* algorithm (Wild, 2017). However, there, the user has to decide before the optimization which one should be used, whereas we switch dynamically between the two. Both algorithms solve the quadratic problem (almost) exactly, which is a good choice for our setting with expensive objective functions. The details of the *BNTR* and *GGTPAR* algorithms are described in Subsection 3.2.5.

If the candidate step s_t is *large enough*, we directly move to the acceptance step. Here, whether a step is large enough is determined based on a cutoff that is relative to the trust-region radius Δ_t^{region}. If the candidate step is too small, we take extra measures to avoid stagnation. If the sample size of the model points is larger than the target sample size n^{target}, we drop n_{stag}^{drop} points, re-fit a vector model, aggregate it, and solve the new trust-region subproblem to get another step size. This process is repeated until the step size becomes large enough or the sample size equals n_{target}. Which points are dropped is again determined by a replaceable component described in Subsection 3.2.6.

If this is not enough to produce a large enough step, we keep dropping n_{stag}^{drop} points, but this time, we replace them with new points. This process is repeated up to n_{stag}^{max} times.

Two things are important to note here: First, solving the trust-region subproblem many times adds some overhead, but this cost is negligible compared to the cost of a single evaluation of the objective function. Second, it seems counterintuitive to drop points instead of simply adding new ones. However, we found that this approach works better in practice. If only new points are added, they have a rather small impact on the model and can, therefore, not avoid stagnation. In our experiments, we found that dropping one point at a time, i.e., $n_{stag}^{drop} = 1$, works best in a serial algorithm.

Once a sufficiently large candidate step s_t has been found or the maximum number of trials for avoiding stagnation has been reached, we move on to the acceptance step. The acceptance step is a replaceable component of *tranquilo* that contributes strongly to the flexibility of our framework.

On an abstract level, the acceptance step looks as follows

$$(x_{t+1}^*, \rho_t) = Accept(x_t^*, s_t, \Delta M_t^s)$$

where x_{t+1}^* is the candidate point for the next iteration, ρ_t is a measure of progress or model quality, and ΔM_t^s is the expected improvement from taking step s_t. x_{t+1}^* can be equal to $x_t^* + s_t$, x_t^* or an entirely different point. ρ_t can either be calculated as in Equation 2.1 or in a different way. Any evaluation of the objective function that happens in the acceptance step will be added to the history and can be used in the next iteration.

This approach nests the classical case where the acceptance step consists of evaluating the objective function at $x_t^* + s_t$ and accepting the step if the improvement is large enough, which often just means larger than zero. Then ρ_t is simply calculated as in Equation 2.1. The implementation of such a simple acceptance step is described in Subsection 3.2.7. However, our approach can also express entirely different methods. Examples are a parallel line-search and speculative sampling, which we will describe in Subsection 4.1.

Given ρ_t and the step size of s_t, we can adjust the trust-region radius for the next iteration. Again, we make the radius adjustment a replaceable component. For our empirical results, we use the radius adjustment rules of the *POUNDERS* algorithm (Wild, 2017), which is described further in Subsection 3.2.8.

If $x_{t+1}^* \neq x_t^*$, we check for convergence of the algorithm. The convergence check is based on the history of function evaluations $\mathcal{H}_t$ as well as the current scalar model M_t^s. This allows for all common convergence criteria, which are either based on absolute or relative improvements in the objective function, absolute or relative step sizes, or the gradient terms of the scalar surrogate model. The exact implementation of the convergence check is again replaceable and described in Subsection 3.2.9

The flexible nature of the *tranquilo* framework allows for quick experimentation and benchmarking of different components that are commonly used in trust-region algorithms. For example, we can easily compare the performance of different model fitting and aggregation strategies, leaving everything else equal. In the traditional literature, these changes would be considered large enough to warrant a new algorithm name (compare, e.g., *DFBOLS* and *POUNDERS*). This flexibility will be extremely useful when looking at extensions for parallelization and noise handling. The basic algorithm as described in this section, will stay virtually unchanged, and only components will be swapped out. The few changes needed in the algorithm itself are the introduction of a few new quantities (e.g., a batch size for parallelization or a noise estimate for noisy problems) that were omitted in the baseline version for simplicity. Moreover, the parallel version of *tranquilo* (see Algorithm **??**) and the noisy version of *tranquilo* (see Algorithm 3) nest the baseline algorithm.

3.2 Implementation of the components

In this section, we provide a detailed description of each component and their different implementations in *tranquilo*. We begin with a discussion of the components of the noise-free serial optimization problem, deferring the discussion of components of the noisy and parallel optimization problem to Subsection 5.3 and 4.1.

3.2.1 Filtering.

At the beginning of each iteration, we get a set of existing points $\mathcal{X}_t^{existing}$ in the neighborhood of the trust-region center x_t^*. These points and their corresponding function evaluations can be used to construct a surrogate model. They are "free" from a computational budget perspective, in the sense that using them for the surrogate model does not incur any additional evaluations of the objective function. However, there can be reasons why not all of those points should be used. First, some of them might be far away from the current trust-region, which might hinder the model from approximating the objective function well locally. Second, some of them might be very close to each other, which can lead the model to overfit certain areas of the trust-region. Third, there might simply be so many points that any newly added point has only a small impact on the model, and therefore, the next candidate point will be very close to the old one.

Filters can address these issues by discarding some of the existing points. More formally, they take the following form

$$\mathcal{X}_t^{filtered} = Filter(\mathcal{X}_t^{existing})$$

Since existing algorithms do not maintain a full history of function evaluations, filtering has no counterpart in the literature. However, the filtering methods we implement are inspired by the traditional goal of producing a well-poised set of model points, i.e., model points with geometric properties that lead to surrogate models with tight error bounds. This topic is discussed in more detail in Subsection 3.2.2.

We implement the following filters:

Keep all. As the name suggests, this filter does not discard any points. We use this filter in our benchmarks for the noise-free and serial case as it yields the best performance in this setting.

Discard all. This filter discards all existing points. Using this filter makes the optimizer slower but, in some cases, more robust, as it uses a new high-quality sample of points in each iteration and is therefore not prone to stagnation. The slowdown is not as severe as one might expect: While each model is now more costly to build, the model quality is also higher, and fewer iterations are needed.

Drop excess. This filter only drops points if more than n^{filter} points are available. If so, we first discard excessive points that are outside the trust-region. We begin with the point farthest from the trust-region center. If all remaining points are inside the trust-region, we look for the two points that are closest to each other and discard the one that is closer to the trust-region center, unless one of the points is the center itself. We repeat this process until only n^{filter} points remain. The idea behind this filter is that we want to have points as far out as possible as long as they are inside the trust-region. This filter is used in our benchmarks for the parallel case with $n^{filter} = 3n^{target}$.

3.2.2 Sampling.

Sampling refers to the process of creating new model points $\mathcal{X}^{model}$ at which the objective function is evaluated in order to construct a surrogate model. In the first iteration, a full sample is created from scratch, and the sample size is set to n^{target}. In most other iterations, sampling only complements a set of existing points, and the sample size might be larger than n^{target}. In all cases, the goal is to create a set of model points with geometric properties that lead to tight error bounds of the surrogate model. Which points are optimal depends on the type of surrogate model used (e.g., linear or quadratic).

We restrict our attention to simple polynomial models that are linear in the parameters. Let n_t denote the number of model points in iteration t, and d the number of coefficients of the model. In this case, the quality of the sample is not directly assessed on the model points $\mathcal{X}^{model} = [x_1, \ldots, x_{n_t}]^T$

but on a design matrix $X \in \mathbb{R}^{n_t \times d}$ that is constructed from the model points given the model class. The design matrix is also known as the matrix of regressors. Note that for all $i = 1, \ldots, n_t$ the model points must be inside the effective trust-region, i.e., $x_i \in R_t \subset \mathbb{R}^p$. Abusing notation slightly, we write $\mathscr{X}^{model} \in R_t^{n_t}$.

In the case of a linear model, we have $d = p + 1$ coefficients, and the construction of the design matrix is as follows

$$D^l(\mathscr{X}^{model}) \equiv X^l = \begin{pmatrix} 1 & x_{1,1} & \ldots & x_{1,p} \\ \vdots & \vdots & \vdots & \vdots \\ 1 & x_{n_t,1} & \ldots & x_{n_t,p} \end{pmatrix}$$

In the case of a quadratic model, there are additional columns for the cross products and square terms, so we have $d = (p+1)(p+2)/2$ coefficients. The design matrix is constructed as follows

$$D^q(\mathscr{X}^{model}) \equiv X^q = \begin{pmatrix} 1 & x_{1,1} & \ldots & x_{1,p} & x_{1,1}^2 & x_{1,1}x_{1,2} & \ldots & x_{1,p}^2 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_{n_t,1} & \ldots & x_{n_t,p} & x_{n_t,1}^2 & x_{n_t,1}x_{n_t,2} & \ldots & x_{n_t,p}^2 \end{pmatrix}$$

There are multiple strands of literature that discuss the optimal sampling of model points. The one that is closest to economics is the one on optimal design (see Pukelsheim (2006) for a comprehensive overview). Optimal design asks the question of how to choose a set of points in the space of potential experiments (which, in our case, is the parameter space) that leads to the most informative data, i.e., data that allows us to estimate parameters of interest with the highest precision. Depending on the goal of the experimenter, different statistical measures of precision are maximized. In the case of a regression model, a frequently used measure is D-optimality. The D-optimal sample is the one whose design matrix minimizes the determinant of the inverse Fisher information matrix, i.e.,

$$\mathscr{X}^{d*} = \underset{\mathscr{X} \in R_t^{n_t}}{\operatorname{argmin}} \det\left(\left[D(\mathscr{X})^T D(\mathscr{X})\right]^{-1}\right)$$

A closely related strand of literature is the one on function approximation. The main difference is that optimal design typically looks at cases where $n_t \geq d$, whereas function approximation looks at the case where $n_t = d$. In this case, the design matrix is square. In the function approximation literature, the sample of choice is called Fekete points. Fekete points are the set of points that

maximize the determinant of the design matrix (see for examples Briani, Sommariva, and Vianello (2012)), i.e.,

$$\mathscr{X}^{f*} = \underset{\mathscr{X} \in R_t^{n_t}}{\operatorname{argmax}} \det\left(D(\mathscr{X})\right)$$

In the case of a square design matrix, the Fekete points are equivalent to the D-optimal sample.[1]

While familiar to economists, neither of the previous approaches extends to the case where $n < d$. Therefore, the literature on trust-region optimizers introduces the concept of Λ-poisedness to measure the quality of samples. This concept is based on Lagrange polynomials and works for over-determined, just-determined, and underdetermined interpolation problems. For a definition and comprehensive treatment of Λ-poisedness, see Conn, Gould, and Toint (2000). While the definition of the measure Λ relies on concepts that are not typically familiar to economists, it has a simple interpretation: Λ^{-1} can be interpreted as the distance a set of model points has to linear dependence, i.e., the smaller Λ is, the more linearly independent the model points are. In the case of just-determined and over-determined interpolation problems, the optimal sample according to Λ-poisedness is equivalent to Fekete points or D-optimal points, respectively.

It is instructive to look at the optimal samples for linear and quadratic models in the case of a spherical trust-region in two dimensions. These samples are shown in Figure 1.

While one might intuitively think that the optimal sample fills the space uniformly, this is not the case. The optimal sample for a linear model consists of points that are uniformly spaced on the sphere, i.e., the boundary of the ball. The same holds for the optimal samples of a quadratic model, except that here, there is one additional point in the center. This pattern carries over to higher dimensions and larger samples. In the case of a cubical-shaped trust-region, the optimal sample also consists of points on the boundary of the cube (and one point in the center for quadratic models), but the spacing is less regular. When moving to higher-order polynomial models, the optimal sample consists of concentric rings (see Briani, Sommariva, and Vianello (2012)).

Calculating optimal samples directly based on Λ-poisedness, D-optimality or the Fekete criterion is expensive. We, therefore, exploit the pattern described above in several of our samplers.

Random hull sampling. This sampler draws points uniformly on the boundary of a spherical or cubic trust-region. When used to complement an existing sample, it does not take the position of existing points into account. The sampler is very fast and provides a good baseline for testing against optimal samplers. Note that even for quadratic models, it is not necessary to sample a point in the center of the trust-region, as the acceptance step from the previous iteration already evaluated the objective function at that point.

1. To see this, note that $\det\left(\left[D(\mathscr{X})^T D(\mathscr{X})\right]^{-1}\right) = \det\left(D(\mathscr{X})^T D(\mathscr{X})\right)^{-1} = \det\left(D(\mathscr{X})\right)^{-2}$.

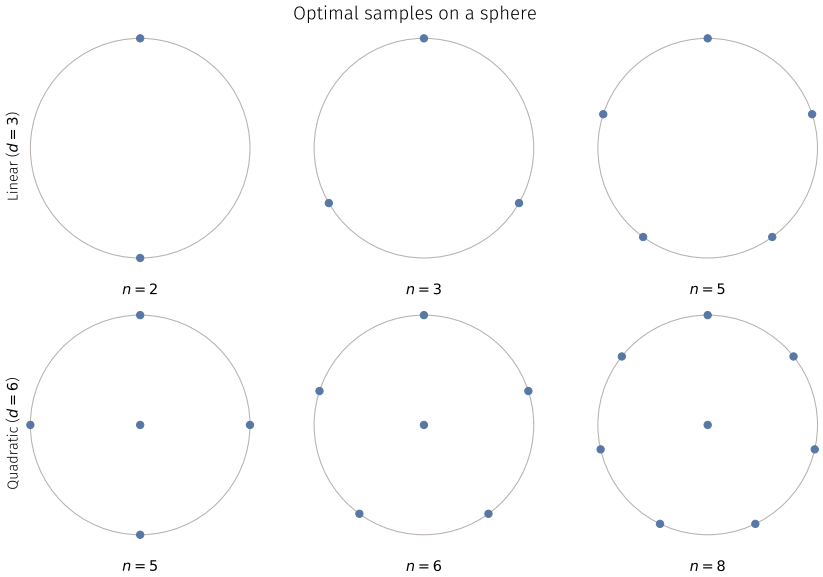


Figure 1. Optimal samples for linear and quadratic models on a ball. The first row shows the optimal samples for linear models, the second row shows the optimal samples for quadratic models. The three columns look at the under-determined, just-determined, and over-determined case. Optimal samples are not space-filling. For linear models, all points lie on the boundary of the ball. For quadratic models, there is one additional point in the center.

Optimal hull sampling. This sampler uses the Random hull sampler to create an initial set of points and refines these points by maximizing the minimal distance between points, i.e.,

$$\underset{\mathcal{X} \in R_t^{n_t}}{\operatorname{argmax}} \left\{ \min \left\{ \|x_i - x_j\| \,:\, i,j \,=\, 1,\ldots,n_t, i \neq j \right\} \right\}$$

To make the problem differentiable, we approximate the minimal distance by a smooth minimum. A smooth minimum of a vector $z = (z_1, \ldots, z_n)$ can be constructed using various approaches. We choose the log-sum-exp function, modified for the minimum-case

$$\text{SmoothMin}(z) \,=\, -\frac{1}{h} \ln \left(\sum_{i=1}^{n} \exp\left(-h z_i\right) \right).$$

The h-parameter determines the hardness of the smooth minimum. As $h \to \infty$, the smooth minimum approaches the true minimum. If used to complement existing points, the optimal hull sampler takes the position of all points into account and positions the new points away from the existing points. This sampler is used by default in *tranquilo* and was used in all benchmarks. It provides a good compromise between sample quality and speed.

Determinant sampler. This sampler creates a D-optimal sample by minimizing the D-optimality criterion using a local optimizer. It is much slower than the optimal hull sampler and can produce lower-quality samples if the optimizer gets stuck in a local optimum. Therefore, this sampler is not used in any of our benchmarks.

3.2.3 Fitting.

Fitting refers to the process of taking a set of model points $\mathscr{X}^{model}$ and corresponding evaluations of the residual function $\mathscr{R}^{model}$ and constructing a surrogate model M^v that approximates the residual function. As is common in the literature, the model points are scaled to the trust-region before fitting. This means that the solution of the trust-region subproblem is always performed over a unit-ball or unit-hypercube. Moreover, scaling increases the numerical stability of the model fitting. We emphasize this by using s instead of x to denote a scaled model point.

We restrict our attention to linear or quadratic models. A linear model for the j-th residual takes the following form

$$M_t^j(s) \;=\; c_t^j + s^T g_t^j \tag{3.1}$$

Where $c_t^j \in \mathbb{R}$ is a scalar intercept term and $g_t^j \in \mathbb{R}^p$ is a vector of slope coefficients. g_t^j is also known as the model gradient. Thus, in total, a linear model has $p+1$ coefficients per residual and we require $p+1$ model points for a just-determined interpolation model.

A quadratic model includes additional terms

$$M_t^j(s) \;=\; c_t^j + s^T g_t^j + \frac{1}{2} s^T H_t^j s \tag{3.2}$$

Here H_t^j is a symmetric matrix of second-order coefficients, which is also known as the model Hessian. Due to the symmetry of H_t^j, the total number of coefficients is $(p+1)(p+2)/2$ and we require $(p+1)(p+2)/2$ model points for a just-determined interpolation model.

If we have more model points than coefficients, the model is over-determined, and instead of solving the interpolation conditions exactly, a least-squares solution is used. If fewer model points than coefficients are available, the model is under-determined, and the remaining degrees of freedom need to be resolved by an additional criterion. Additional criteria are usually based on the absolute values or norm of the model coefficients. Methods differ according to two main dimensions: First, whether they penalize all coefficients or only the coefficients in H_t^j. And second, whether they penalize the magnitude of the coefficients or the magnitude of the change in coefficients between two iterations. To capture the first dimension, we implement different fitting methods. To capture

the second one, we use a residualization approach that can be used in combination with any fitting method.

Residualization means that on the left-hand side of the interpolation conditions, we do not use $\mathscr{R}_t^{model}$ directly. Instead, we subtract the predicted residuals of the previous model M_{t-1}^r, evaluated at the current model points $\mathscr{X}_t^{model}$ from the residuals. Thus, the coefficients of the fitted model only capture the difference between the previous and the current model, and any penalization that might be done by the fitting method only penalizes the change in coefficients. After fitting the model, the coefficients of the previous model can be added back to produce a model that approximates the current residual function.

OLS fitting. OLS fitting solves overdetermined models by minimizing the squared norm of the residuals. For just-determined models, this is equivalent to solving the interpolation conditions exactly. For underdetermined models, there are multiple solutions to the interpolation conditions. From those solutions, the solution where the Euclidean norm of all coefficients is smallest is chosen. If used in combination with residualization, the solution with the smallest change in coefficients (in Euclidean norm) is chosen.

Hessian-norm fitting. This fitting method is equivalent to OLS fitting for over and just-determined models. For underdetermined models, it penalizes the Frobenius norm of the Hessian coefficients. This approach is used by Wild (2008) and motivated by theoretical results that guarantee an approximation quality of the quadratic model (Larson, Menickelly, and Wild, 2019). If used in combination with residualization, the penalty is only applied to the change in Hessian coefficients between two iterations. This approach was first introduced by Powell (see Powell (2009) and Powell (2006)) and is also used in *POUNDERS* (Wild, 2017) and several other algorithms (Larson, Menickelly, and Wild, 2019).

Mixed fitting. Motivated by the comparable success of OLS-fitting and Hessian-norm-fitting in our benchmarks, we also implement a fitting method that combines the two approaches smoothly. Instead of only penalizing Hessian coefficients or penalizing all coefficients equally, we introduce the possibility of weighted penalization that differs across the intercept, gradient terms, and Hessian terms. We use this fitting method by default for underdetermined fitting problems and find that we get the best performance if we put a weak penalty on the intercept, a medium penalty on the gradient terms, and a strong penalty on the Hessian terms. The exact weights are not interpretable and were set empirically by tuning the algorithm against a benchmark set. The fitting method uses standard OLS-fitting after scaling the columns of the design matrix. After the fitting, we rescale the coefficients to undo the effects of rescaling the data. For over- and just-determined problems, this fitting method is equivalent to standard OLS-fitting.

Ridge fitting. An alternative approach to the above three fitting methods is ridge regression. Ridge regression performs an ℓ_2-regularization of the coefficients by adding a penalty to the objective function of a least-squares regression

$$\min_{\Theta \in \mathbb{R}^{k \times d}} \sum_{i=0}^{n_t} \|M^v(\tilde{x}_i; \Theta) - r(x_i)\|^2 + \lambda \|\Theta\|^2 \tag{3.3}$$

where Θ are the coefficients of the regression problem, and the constant λ is a penalty term that controls the shrinkage of coefficients, which leads to relatively smaller estimates for coefficients with low explanatory power. The big difference between ridge regression and the fitting methods discussed above is that the penalty has an effect even for over-determined models. While this could be attractive in the presence of noise, it introduces a practical problem: The penalty parameter λ has to be set, for which we did not find an adequate solution that performed well across all benchmarks.

3.2.4 Aggregation.

Aggregation refers to the process of converting a vector model M_t^v, that approximates the residual function $r(x)$, into a scalar model M_t^s, that approximates the objective function $f(x)$. The scalar model is used to solve the trust-region subproblem to produce a candidate point s_t. While vector models might be linear, we only consider aggregation methods that result in quadratic scalar models. This is because linear scalar models are not capable of having internal optima, which makes them unsuitable for trust-region optimization. The choice of aggregation method depends on the type of residual model (linear or quadratic) and the type of objective function (scalar or least-squares). We implement aggregation methods for three cases:

Least-squares objective, linear residual models. With linear vector model M_t^v, we follow *DF-OLS* by building the scalar model M_t^s through substitution of r^j by M_t^j (Equation 3.1) in the definition of the full objective function DET-LS

$$M_t^s(s) \equiv \sum_{j=1}^{k} M_t^j(s)^2 = c_t^s + s^T g_t^s + \frac{1}{2} s^T H_t^s s$$

Where

$$c_t^s \equiv \sum_{j=1}^{k} (c_t^j) \in \mathbb{R} \tag{3.4}$$

$$g_t^s \equiv 2 \sum_{j=1}^{k} c_t^j g_t^j \in \mathbb{R}^p \tag{3.5}$$

$$H_t^s \equiv 2 \sum_{j=1}^{k} (g_t^j)(g_t^j)^T \in \mathbb{R}^{p \times p} \tag{3.6}$$

We define the gradient of M_t^s as $g_t^M = \frac{d}{ds} M_t^s$, which can be derived as

$$g_t^M(s) = g_t^s + H_t^s s \tag{3.7}$$

Least-squares objective, quadratic residual models. With quadratic residual models, *POUNDERS* obtains an aggregate model using a "full Newton" approach. The full Newton model approximates the scalar model obtained by direct substitution of the residual functions by a second-order Taylor expansion around the current candidate point x_t^*

$$M_t^s(s) \equiv \sum_{j=1}^{k} (M_t^j(s))^2 \approx c_t^s + g_t^s s^T + \frac{1}{2} s^T H_t^s s \tag{3.8}$$

where c_t^s and g_t^s are defined as in 3.4 and 3.5, respectively, and

$$H_t^s \equiv 2 \sum_{j=1}^{k} \left((g_t^j)(g_t^j)^T + H_t^j c_t^j \right)$$

An alternative approach is implemented in *DFBOLS* (Zhang, Conn, and Scheinberg, 2010), where the second-order term of the scalar model is regularized based on cut-offs on the intercept and the linear terms. The regularization is designed to provide fast local convergence for problems with sparse residuals (Zhang, Conn, and Scheinberg, 2010). This approach could be implemented in *tranquilo* in the future.

Scalar objective, quadratic "residual" models. In the case of a scalar objective function, the residual function is simply the objective function, and the aggregation method is the identity function. Using the term aggregation here is simply an abstraction that allows us to use the same algorithmic framework for both cases.

3.2.5 Subsolvers.

After obtaining a scalar model M_t^s, we solve the trust-region subproblem to obtain a candidate step s_t. The model is already scaled such that the subproblem is solved over the same space in each iteration, which is either a unit-ball or a unit-hypercube. More formally, we solve one of the following problems

$$\min_{\tilde{s} \in \mathbb{R}^p} M_t^s(\tilde{s}) \text{ s.t. } \|\tilde{s}\| \leq 1 \tag{SP-Ball}$$

$$\min_{\tilde{s} \in \mathbb{R}^p} M_t^s(\tilde{s}) \text{ s.t. } \tilde{s} \in [-1, 1]^p \tag{SP-Cube}$$

After solving the subproblem, the resulting vector $\tilde{s}$ is rescaled with the radius of the effective trust-region to obtain the candidate step s_t.

The literature on subproblem optimizers is extremely well-developed, and we do not innovate in this area. Traditionally, subproblem solvers only look for approximate solutions in order to save

computational resources. However, in a setting with expensive objective functions, solving the subproblem precisely incurs only a negligible overhead that is outweighed by the benefits of a precise solution. For solving the Problem SP-Ball, we use the *GQTPAR* algorithm. For solving the Problem SP-Cube, we use *BNTR*. Both algorithms are also used by *POUNDERS* (Wild, 2017). We provide numba-accelerated Python reimplementations of both algorithms. Our implementations are described further in Appendix B.

3.2.6 Dropping Points.

To avoid stagnation, there are two situations in which we drop points: In the while loop starting in Line 22, we are in a situation where the sample is larger than the target sample size and drop points without replacing them. In the while loop starting in Line 31, we have reached the target sample size and replace each dropped point with a new one. In both cases, we use the same dropping algorithm which is also used by the drop-excess filter described in Subsection 3.2.1.

3.2.7 Acceptance decision.

The way we formalize the acceptance step in *tranquilo* plays a key role in making *tranquilo* a flexible algorithmic framework for trust-region optimization. Formally, the acceptance step looks as follows

$$(x^*_{t+1}, \rho_t) = Accept(x^*_t, s_t, \Delta M^s_t) \tag{3.9}$$

where x^*_{t+1} is the candidate point for the next iteration, ρ_t is a measure of progress or model quality, and ΔM^s_t is the expected improvement from taking step s_t.

Within these boundaries, many different implementations of acceptance steps are possible. Traditionally, ρ_t is calculated as the ratio of actual and expected improvement (see Equation 2.1), and x^*_{t+1} is either the candidate point $x^*_t + s_t$ or the current point x^*_t. In some algorithms, x^*_{t+1} can also be a model point if it yields an improvement over both the candidate point and the current point (see, for example, Cartis, Fiala, et al. (2019)). Typically, the acceptance step comprises only one new objective function evaluation at $x^*_t + s_t$.

In *tranquilo*, the acceptance step can calculate ρ_t in any way that is useful for the radius management and can use any number of objective function evaluations to create x^*_{t+1}. While we stick to traditional approaches for the serial and noise-free case, our extensions to parallel and noisy settings mainly consist of modifications to the acceptance step. Those extensions are described in Subsection 4.1.2 and 5.3.5.

Accept classic. In this acceptance step, ρ_t is calculated as in Equation 2.1 and x^*_{t+1} is either $x^*_t + s_t$ or x^*_t. The candidate point is accepted if it yields any improvement over the current point.

3.2.8 Trustregion radius adjustment.

We base the implementation of the trust-region radius adjustment step on the radius adjustment rules of Wild (2017), which is given by

$$\Delta_{t+1}^{region} = \begin{cases} \min\{\gamma^{inc}\Delta_t^{region}, \Delta^{max}\} & \text{if } \rho_t \geq \rho^{inc} \text{ and } s_t \geq c^{ls}\Delta_t^{region} \\ \gamma^{dec}\Delta_t & \text{if } \rho_t < \rho^{dec} \\ \Delta_t & \text{otherwise} \end{cases} \tag{3.10}$$

As we can see from Equation 3.10, the updates to the trust-region radius depend on model performance, measured by ρ_t, and the length of the step s_t. Only if both are large, the trust-region radius is increased by a factor γ^{inc}. Here, a high ρ_t indicates that the model is good enough that we can afford a larger radius. A large step length indicates that the solution lies outside the current trust-region, and we would actually benefit from a larger trust-region. What counts as a large enough ρ_t is determined by a constant cutoff ρ^{inc}. As in *POUNDERS* (Wild, 2017), we bound the trust-region radius by a constant Δ^{max}.

On the other hand, if the ratio of actual to expected improvement falls below a threshold $\rho^{dec} \leq \rho^{inc}$, we shrink the trust-region radius by a factor $\gamma^{dec} \leq \gamma^{inc}$.

For the values of ρ_t between cut-off values ρ^{dec} and ρ^{inc}, we leave the trust-region radius unchanged. Similarly, if $\rho_t > \rho^{inc}$ but the step-length is small $s_t < c^{ls}\Delta_t^{region}$, we also leave the trust-region radius unchanged.

In *tranquilo*, we use the values $\rho^{dec} = \rho^{inc} = 0.1$ for the cut-offs on the ratio ρ_t. For the expansion and shrinkage factors of the trust-region radius, we use the values $\gamma^{inc} = 2$ and $\gamma = 0.5$. To identify large candidate steps, we use the value of $c^{ls} = 0.5$ for the relative step length. Finally, for Δ^{max}, we use the value of 10^6. All of these values are taken from the TAO implementation of *POUNDERS* (Dener et al., 2021).

3.2.9 Convergence and stopping criteria.

We use common convergence criteria in Line 35 of Algorithm 1 based either on absolute or relative improvements in the objective function, absolute or relative step sizes, or the linear terms of the scalar surrogate model. Specifically, the algorithm stops at iteration t if $Converged(\mathcal{H}_t, M_t^s, x_t^*, x_{t+1}^*)$ in Line 35 evaluates to *True*. This happens if any of the following conditions are satisfied

$$
\begin{aligned}
|f(x_t^*) - f(x_{t+1}^*)| &\leq \epsilon^{fatol} \\
|f(x_t^*) - f(x_{t+1}^*)|/|f(x_t^*)| &\leq \epsilon^{frtol} \\
\|g_t^M(x_{t+1}^*)\| &\leq \epsilon^{gatol} \\
\|g_t^M(x_{t+1}^*)\|/|f(x_{t+1}^*)| &\leq \epsilon^{grtol} \\
\|x_{t+1}^* - x_t^*\| &\leq \epsilon^{xatol} \\
\|x_{t+1}^* - x_t^*\|/\|x_t^*\| &\leq \epsilon^{xrtol}
\end{aligned}
$$

These convergence criteria are taken from the *POUNDERS* implementation, described in Dener et al. (2021). Note that g_t^M is the gradient of the scalar model M_t^s, as defined in Equation 3.7.

3.3 Benchmarking

The ideal way to evaluate the performance of an optimization algorithm would be to run it on a large set of real-world problems and compare its performance to other optimizers. However, this approach is not feasible for several reasons. First, for interesting real-world problems, the exact solution is typically unknown. Second, the real-world problems we are interested in are too costly to be used in a benchmark. Third, there are no standard sets of real-world problems, so our results would not be comparable to other results in the literature.

For these reasons, it is common to evaluate optimization algorithms on standardized sets of benchmark problems with known solutions. These problems are designed to be representative of real-world problems and to include features that are challenging for optimization algorithms. However, they are fast to evaluate, so the benchmark can be run in minutes or hours instead of days or weeks. A complete benchmark is defined in terms of a set of problems, a set of solvers, and a convergence test (Dolan and Moré (2002)).

Throughout this paper, we use modified versions of the Moré-Wild benchmark set (Moré and Wild (2009)) to evaluate the performance of our algorithms. This benchmark set contains 53 non-linear least squares problems with known solutions. These test cases are constructed based on 22 functions originally derived from the CUTEr Problems (Gould, Orban, and Toint (2003)). The objective functions are twice continuously differentiable, but we do not make use of the derivatives in our benchmarks. The parameter dimensions p vary between 2 and 12, where the median dimension is 7. The dimension of the least squares residuals k is between 2 and 65. Only three of the 53 problems have local minima that are not global minima. These are based on the Freudenstein and Roth function and the Brown almost-linear function. The remaining 50 problems each have a unique minimum.

The Moré-Wild benchmark set is standard in the recent literature on derivative-free optimization. Among others, it has been used to benchmark *POUNDERS* (Wild (2017)), *DFOGN* (Cartis and Roberts (2019)), and *DFOLS* (Cartis, Fiala, et al. (2019)).

The benchmark set plays an important role not only in measuring the final performance of the algorithm but also in tuning the algorithm's hyperparameters during development. In order to avoid overfitting the tuning parameters to the benchmark set and to improve the robustness of our conclusions, we extend the benchmark set with randomly generated problems. For each of the 53 problems, we generate four additional problems by drawing a new vector of start parameters in the neighborhood of the original start parameters. The neighborhood is defined by multiplying the original start parameters with 0.9 or 1.1. In the case of parameter values smaller than 1, we switch to additive perturbations by adding and subtracting 0.1. The new start parameters are drawn uniformly from the neighborhood. If the objective function is undefined at the new starting values, we tighten the neighborhood until we find a valid starting point.

To measure the performance of different algorithms, we need a convergence test. Importantly, a convergence test is only based on the history of function values of each optimizer and the known solution of the problem. It is independent of the algorithm's internal convergence criteria. We use the following convergence test, as proposed by Moré and Wild (Moré and Wild (2009)), to test whether algorithm a solved problem i

$$\frac{f_i(x_{ia}^*) - f_i^*}{f_i(x_{i0}) - f_i^*} \leq \tau \tag{3.11}$$

where $\tau > 0$ is a tolerance level, x_{i0} is the vector of start parameters, f_i^* is the known minimum of the objective function, and $f_i(x_{ia}^*)$ is the lowest objective function value obtained by the optimizer. Note that x_{ij}^* can be any point that has been tried out by algorithm j. For noise-free problems, we set $\tau = 10^{-3}$.

Once we have the convergence test to decide whether an algorithm solved a problem, we need a way to measure the computational budget the algorithm needed until it found a solution. The computational budget can also be interpreted as the runtime until solution. Since we are interested in applications where the objective function is expensive, meaning that, by assumption, the algorithm will spent most of its runtime on evaluating the objective function, we use the number of function evaluations as the measure of the computational budget. Using walltime instead would mostly measure how much work is done inside the algorithm itself because all objective functions in the benchmark set are very fast to evaluate. Using the number of function evaluation is common practice in the literature on derivative-free least-squares optimization (see for example Wild (2017) and Cartis, Fiala, et al. (2019)).

The standard way of visualizing the performance of a set of solvers on a benchmark set are *performance profiles* (Moré and Wild (2009)), which are also known as *profile plots*. Performance profiles show the share of solved problems on the y-axis. On the x-axis, they show a normalized measure of the computational budget. Normalized here means that the number of function evaluations each algorithm needed to solve a given problem is divided by the runtime that the fastest algorithm needed to solve the problem. This makes performance profiles useful even for benchmark sets that contain problems with very different difficulty levels. Without normalization, the performance profile would be dominated by the hardest problems. The x-axis of performance profiles starts at 1.

The y-value each algorithm achieves at 1 is the share of problems for which this algorithm was the fastest.

Figure 2 shows the performance profiles for the least-squares version of *tranquilo* and compares it against *DFO-LS* and *POUNDERS*.

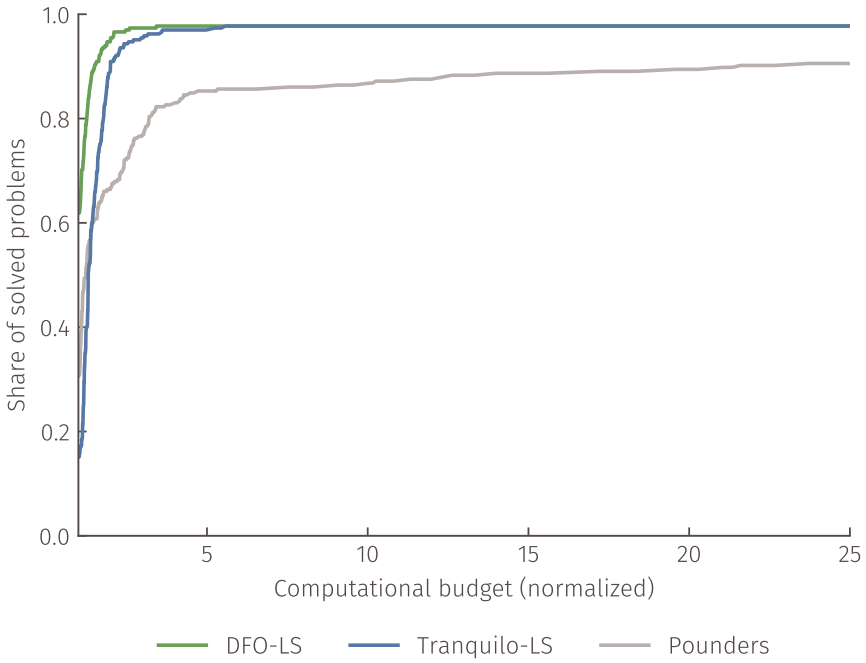


Figure 2. Comparison of least-squares optimizers on an augmented Moré-Wild benchmark set. The y-axis shows the share of problems solved. The x-axis shows the normalized computational budget. The computational budget is measured in terms of objective function evaluations needed by the optimizers. Normalized means that the number of function evaluations each algorithm needed to solve a given problem is divided by the number of function evaluations the fastest algorithm needed to solve that problem.

Both *DFO-LS* and *tranquilo* solve the same number of problems. In most problems, *DFO-LS* is slightly faster than *tranquilo*. *POUNDERS* is slower than the other two on most problems. Moreover, it fails to solve some problems to the required level of precision. This is in line with results by Cartis, Fiala, et al. (2019) who suspect that the lack of precision is related to the minimal trust-region radius *POUNDERS* uses.

Figure 3 shows the performance profiles for the scalar version of tranquilo and compares its performance against *BOBYQA* implementations from NlOpt and the Numerical Algorithms Group (NAG) as well as *Nelder-Mead* implementations from NlOpt and SciPy.

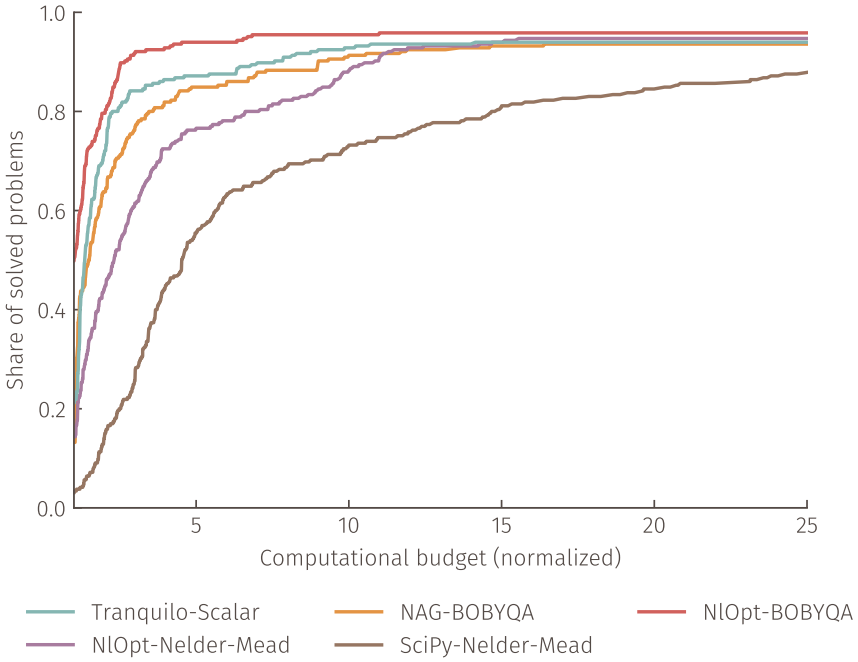


Figure 3. Comparison of scalar optimizers on an augmented Moré-Wild benchmark set. The *y*-axis shows the share of problems solved. The *x*-axis shows the normalized computational budget. The computational budget is measured in terms of objective function evaluations needed by the optimizers. Normalized means that the number of function evaluations each algorithm needed to solve a given problem is divided by the number of function evaluations the fastest algorithm needed to solve that problem.

The fastest and most robust optimizer is the NlOpt implementation of *BOBYQA*. The slowest and least robust optimizer is the SciPy implementation of *Nelder-Mead*. All other algorithms solve slightly fewer problems than the NlOpt implementation of *BOBYQA*. Among them, *tranquilo* is the fastest, followed by the NAG implementation of *BOBYQA* and the NlOpt implementation of *Nelder-Mead*. Generally, the derivative free trust-region optimizers seem faster and more robust than the direct search methods.

Figure 4 combines the two cases and compares scalar and least-squares algorithm in a single plot. The main purpose of this plot is to show that the least-squares algorithms indeed outperform similar scalar algorithms when applicable.

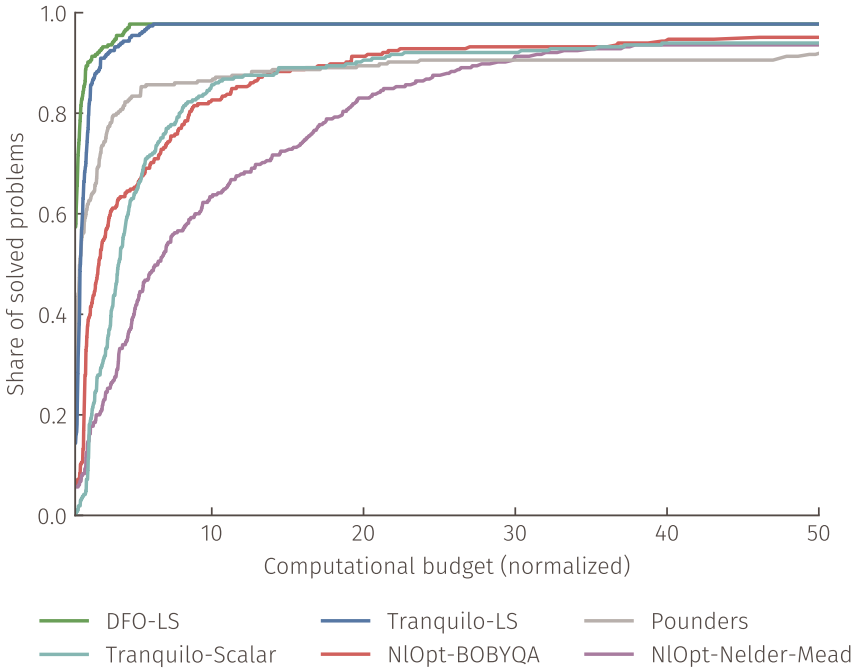


Figure 4. Comparison of scalar and least-squares optimizers on an augmented Moré-Wild benchmark set. The *y*-axis shows the share of problems solved. The *x*-axis shows the normalized computational budget. The computational budget is measured in terms of objective function evaluations needed by the optimizers. Normalized means that the number of function evaluations each algorithm needed to solve a given problem is divided by the number of function evaluations the fastest algorithm needed to solve that problem.

The combined plot shows a clear separation of several groups of algorithms: Least-squares algorithms that use linear residual models and aggregate them into quadratic scalar models are clearly faster than all other algorithms. *POUNDERS*, as the only least-squares algorithm that uses a quadratic residual model, is faster than scalar optimizers but cannot solve all problems to the required level of precision. The scalar optimizers can again be split into two groups: The two *BOBYQA* implementations (i.e., model-based trust-region optimizers) are faster than the NlOpt implementation of *Nelder-Mead* (i.e., a direct search method). The SciPy implementation of *Nelder-Mead* is omitted because it is much slower than the other algorithms (see Figure 3).

4 Parallelization

Tranquilo is designed for objective functions f that cannot easily be parallelized. This means that if parallel hardware is available, the parallelization should be done on the algorithm level.

When designing a parallel algorithm, the focus shifts from minimizing the number of objective function evaluations to minimizing the number of *batches*, where each batch consists of n^{batch} objective function evaluations that can be done in parallel.

For instance, assume that n^{batch} is equal to four and we require seven function evaluations. This example is depicted in the following illustration:

$$\left\{\left[\underbrace{f(x_1), f(x_2), f(x_3), f(x_4)}_{\text{Batch 1}}\right], \left[\underbrace{f(x_5), f(x_6), f(x_7)}_{\text{Batch 2}}\right]\right\}$$

If all seven function evaluations are independent, we can perform them in two batches. Further, the second batch is not full. If we do not derive utility from idle resources, an additional evaluation in the second batch would come as a "free lunch." The goal of our parallelization approaches in tranquilo is therefore to do as many function evaluations as possible in parallel and to find good ways to use up any "free" evaluations.

In most cases, the batch size (n^{batch}) is equal to the number of cores that are available to the optimizer.

The basic version of *tranquilo* as described in Algorithm 1 already creates some situations in which the objective function needs to be evaluated on multiple points and there are no dependencies between these evaluations: In the first iteration, the objective function is evaluated at every point in the initial sample which contains at least $p + 1$ points. In all subsequent iterations, the objective function is evaluated at all newly sampled points. Of course, the parallel version of tranquilo exploits these situations. However, in most of these cases, the number of function evaluations required are not multiples of the batch size and therefore, "free" evaluations are left on the table. Moreover, there are several situations in which just one function evaluation is required. This is for example the case when points are replaced to avoid stagnation (Line 22 of Algorithm 1) and in the acceptance step. The parallel version of tranquilo exploits most of these situations and fills up the "free" function evaluations with different strategies.

4.1 Adding parallelization to tranquilo

Almost all of the changes required to add parallelization to tranquilo are done by switching out components. The only exception is the sampling step in each iteration. Here, instead of passing n^{target} as target sample-size into the samplers (see Line 13), we pass in a target sample size that makes sure that the number of newly sampled points is a multiple of n^{batch}. Moreover, the acceptance step now depends on the batch size.

4.1.1 Filtering.

As the usage of these "free" evaluations potentially leads to many more available points in the history, we observed that using no filter leads to worse benchmark results compared to using the *Drop excess* filter. The *Drop excess* filter is described in Subsection 3.2.1 and activated by default when $n^{batch} > 1$. In the parallel benchmarks we set $n^{filter} = 3n^{target}$; see Figure 7 for reference.

4.1.2 Acceptance Step.

During the acceptance step, we determine the new candidate point x^*_{t+1} and a measure of model quality ρ_t. A typical serial acceptance step requires a single function evaluation at the candidate point $x^*_t + s_t$ (see Subsection 3.2.7). In the parallel case, this leaves us with many "free" evaluations, which we can use to improve efficiency.

In the parallel acceptance step, we invest one evaluation in the candidate point, which leaves us with $n^{batch} - 1$ available evaluations to fill up the batch. We use these in two ways. First, we check whether the candidate point lies at the trust-region border. If so, we interpret it as a signal that the direction of the step is good, but the step size might be too small. We thus sample points on the line that goes through the current best point and the candidate point. We call this a line search. Second, we try to predict at which points the objective function needs to be evaluated in the next iteration and perform the evaluations. We call this speculative sampling.

Line Search. Consider the illustration in Figure 5. For the line search, we sample points on a line starting at the current best point (the center point of the circle in the illustration) and going through the candidate point (the red point in the illustration). Formally, the line is given by

$$Line(\alpha) = x^*_t + \alpha s_t$$

where $Line(1)$ equals the candidate point. For values $\alpha < 1$, $Line(\alpha)$ represents points inside the trust-region, while for $\alpha > 1$, $Line(\alpha)$ represents points outside of the trust-region. Since we believe the step was too small, we want to sample outside of the trust-region. In one iteration, the trust-region radius can increase by a maximum of 2. To simulate a continuous maximal increase of the radius, we sample a maximum of three points on the line with $\alpha = 2, 4, 8$, respectively, depending on the number of "free" evaluations (the blue points in the illustration).

If the search direction of the candidate step is good, a line search can dramatically increase the speed of the algorithm, allowing us to go as far as 2^3 times the initial trust-region radius, which translates to jumping ahead three iterations of the algorithm. If any of the line search points is better than the current best point, we accept it.

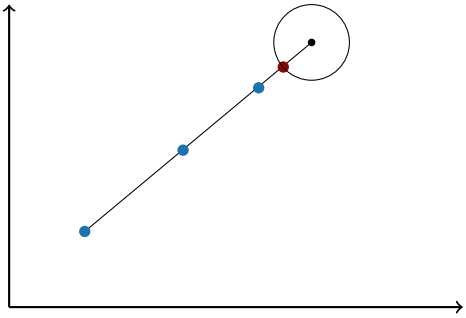


Figure 5. Illustration of the line search. The candidate point is shown in red. The black dot shows the current center. The blue dots show the line search points. The line search points are all on a line that goes through the current best point and the candidate point. The spacing is at 2, 4, and 8 times the current current trust-region radius.

Speculative Sampling. Consider the illustration in Figure 6. For the speculative sampling, we assume that the candidate point will be accepted (the red point in the illustration). In this case, we can use any "free" evaluations to sample points around the candidate, as these points will be required in the next iteration. We do not know, however, how the radius will change. After empirical testing, we found that setting the radius of the region from which we draw the speculative sampling to 0.75 times the current trust-region radius results in the best benchmark performance. The speculative sample points are shown in blue in the illustration.

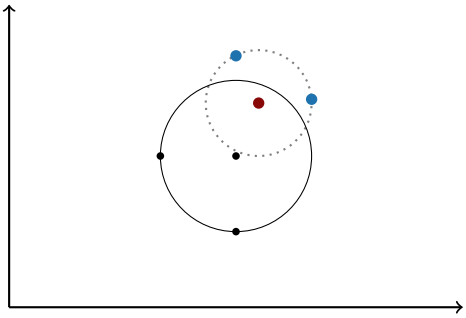


Figure 6. Illustration of the speculative sampling. The candidate point is shown in red. The black dots show a hypothetical sample of existing points that would be available in the next iteration if the candidate point was accepted. The blue dots show the speculative sample. The points are sampled in the same way they would be sampled in the next iteration if the candidate point was accepted and the radius was 0.75 times the current trust-region radius.

Implementation of the Parallel Acceptance Step. If enough "free" function evaluations are available, we combine both, the line search and speculative sampling in our parallel acceptance step. For an efficient combination, we first calculate the line-search points and already take them into account as existing points when creating the speculative sampling. Of course, the function evaluations

on both, the line-search points and the speculative sample are done in parallel, together with the function evaluation at the candidate point. The parallel acceptance step is shown in Algorithm 2.

Algorithm 2: Parallel acceptance step

Input: The current parameter vector x_t^*, the candidate step s_t, the expected improvement ΔM_t^s, the effective trust-region R_t, the history $\mathcal{H}_t$, and the batch size n^{batch}.

if $n^{batch} = 1$ **then**
  return $AcceptClassic(x_t^*, s_t, \Delta M_t^s)$
else
end
if $x_t^* + s_t$ *is on the border of* R_t **then**
  Calculate the number of available line search points: $n^{ls} = \min\{n^{batch} - 1, 3\}$
  Sample n^{ls} points on a line: $\mathcal{X}_t^{ls} = \{x_t^* + 2^i s_t : i = 1, \ldots, n^{ls}\}$
else
  $n^{ls} = 0$ and $\mathcal{X}_t^{ls} = \{\}$
end
Calculate number of speculative sampling points: $n^{speculative} = n^{batch} - 1 - n^{ls}$
Define a speculative sampling region: $R_t^{speculative}$ with the same center as R_t, and a radius of 0.75 times that of R_t
Scan the history for existing points $\mathcal{X}_t^{existing} = \{x \in \mathcal{H}_t : x \in R_t^{speculative}\}$
Sample speculative points: $\mathcal{X}_t^{speculative} = Sample(\mathcal{X}_t^{ls} \cup \mathcal{X}^{existing}, R_t^{speculative}, n^{speculative})$
Compute $\rho_t = (f(x_t^*) - f(x_t^* + s_t))/\Delta M_t^s$
Compute $x_{t+1}^* = \text{argmin}\{f(x) : x \in \{x_t^* + s_t\} \cup \mathcal{X}_t^{ls} \cup \mathcal{X}_t^{speculative}\}$
return (x_{t+1}^*, ρ_t)

4.2 Benchmarking

The performance-profiles have to be adjusted for the parallel case, as the number of objective evaluations does not provide a good measure of runtime anymore. Instead, we use the number of batch evaluations to measure the computational budget. This reflects our assumption that in the parallel case the evaluation of a batch takes as much time as the evaluation at a single point.

Figure 7 shows the benchmark results of our parallel algorithm on the Moré and Wild (2009) benchmark set. We compare the parallel least-squares version of *tranquilo*, for batch sizes 2, 4, and 8, to the serial version of *tranquilo* and the *DF-OLS* algorithm.

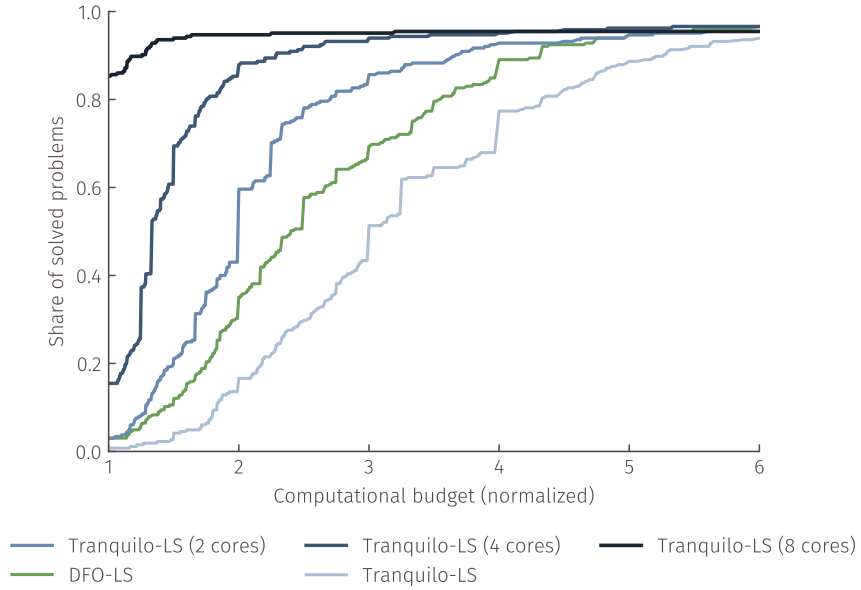


Figure 7. Comparison of parallel and serial least-squares optimizers on an augmented Moré-Wild benchmark set. The *y*-axis shows the share of problems solved. The *x*-axis shows the normalized computational budget. The computational budget is measured in terms of batches of objective function evaluations needed by the optimizers. Normalized means that the number of batches each algorithm needed to solve a given problem is divided by the number of batches the fastest algorithm needed to solve that problem.

As in Subsection 3.3, the *y*-axis denotes the share of solved problems, while the *x*-axis denotes the multiple of the minimal number of *batches* needed to solve the problem. This implies that the intercept can be interpreted as the share of problems that the respective algorithm was able to solve the fastest.

The serial version of *tranquilo* (lightest blue) is slower than the *DF-OLS* algorithm (green), as was also seen in the least-square benchmark (see Figure 2). The parallel versions of *tranquilo*, however, dominate the *DF-OLS* algorithm for given batch sizes. In particular, when using a batch size of 8, *tranquilo* is the fastest algorithm for roughly 85% of the problems. In some sense, this comes as no surprise, as there are multiple effects playing a role when using a larger batch size. First, the sample sizes will be larger, and second, we can fully utilize the combination of line search and speculative sampling.

5 Noisy optimization

As described in Section 2.2, the main challenge for trust-region optimizers in the case of noisy objective functions is to determine how often the objective function should be evaluated at each point. Multiple evaluations at the same points are necessary in order to average out the noise to a level that allows the optimizer to make progress. *DFO-LS* places this burden on the user. *ASTRO-DF* determines the sample size adaptively by adding evaluations until an estimated standard error falls under a fixed factor of the squared trust-region radius.

Tranquilo takes an entirely different approach that recognizes that the effects of noise are similar to the effect of approximation error in the surrogate model –which is handled very well by trust-region optimizers. *Tranquilo* therefore introduces a new measure of model quality, ρ^{noise}, that can be used to adjust the number of function evaluations used to construct surrogate models. Moreover, we employ statistical power analysis to determine the number of function evaluations required to make an acceptance decision.

The structure of this section is as follows: Subsection 5.1 discusses the effects of setting sub-optimal sample sizes and why determining optimal sample sizes ex-ante is hard. Subsection 5.3 describes the changes to the core algorithm framework that are necessary to make *tranquilo* robust to noise, as well as the implementation of new components for noisy optimization. Subsection 5.4 compares the noise-robust version of *tranquilo* against different configurations of *DFO-LS* on a noisy version of the Moré-Wild benchmark set.

5.1 The importance of sample sizes

To make things precise, we use the following notation: m_t^{model} denotes the number of repeated function evaluations at each model point in iteration t. m_{t1}^{accept} and m_{t2}^{accept} are the number of repeated function evaluations at the current x and the candidate point in the acceptance step of iteration t. For convenience, most algorithms for noisy optimization set all three numbers equal, even though they are conceptually quite different. In *tranquilo*, we therefore keep the distinction, and since each number of repetitions is set adaptively, they do not generally coincide. Although the number of repetitions (symbolized by the letter m) will often be called sample size in the following sections, it is not to be confused with the sample size as used in the earlier sections of this paper (symbolized by the letter n), which measures the number of distinct x-vectors used for building a surrogate model. If the distinction is not clear from the context, we will use the term number of repetitions.

If m_t^{model} is too small, the surrogate model will be estimated imprecisely, even if the trust-region radius is chosen appropriately and a quadratic model can approximate the true objective function well. This means that the candidate points obtained from minimizing the surrogate model have low quality, and therefore, the measure of model quality ρ will be low in many iterations. If no further measures are taken, the radius is decreased until it collapses to zero, and the algorithm stagnates. On the other hand, if the sample size is too large, the algorithm will become prohibitively expensive.

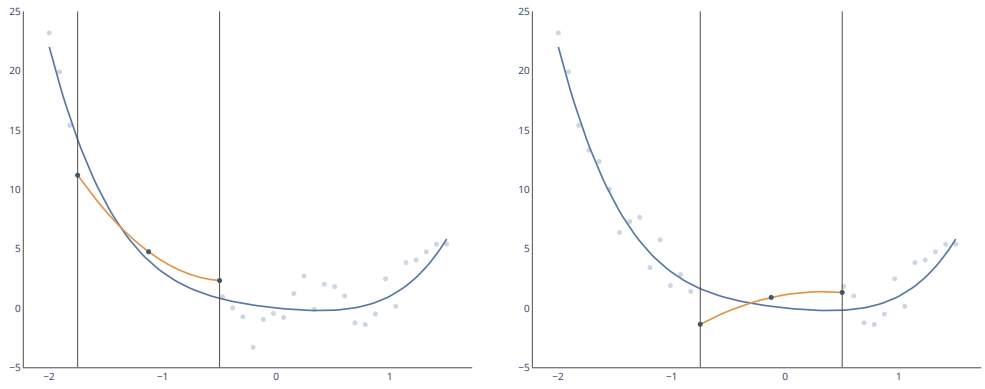
A similar effect occurs in the acceptance step: If the sample size is too small, the acceptance decision will be based on noisy information. It becomes possible that candidates that are worse than the current point in expectation are accepted due to lucky draws and, conversely, it can happen that very good candidates are rejected. Moreover, ρ becomes noisy and the radius adjustment erratic.

To talk about noisy and noise-free function evaluations and residuals, respectively, we use the following conventions: $f(x, \xi_j)$ is the j-th noisy realization of the objective function at x and $\mathbb{E}f(x, \xi)$ is the expectation of the objective function at x. $\overline{\mathscr{F}}_t^{model}$ is used to denote the average of all function evaluations at the model points. Analogous notation is used for the residual function r.

Figure 8 illustrates why it is hard to pick optimal sample sizes. We focus on m^{model}, but very similar arguments apply to m^{accept}. The left and right plot show the same objective function. The vertical lines mark trust-regions bounds. In both plots, the trust-region radius is the same, but the centers differ. In each trust-region, we plot a D-optimal sample (x_1, x_2, x_3). At each point, we observe one noisy function evaluation $f(x_1, \xi_1)$, $f(x_2, \xi_1)$ and $f(x_3, \xi_1)$. The realizations of the random term ξ are the same in both plots.

In the left plot, the trust-region is in a steep area of the objective function. While the effect of noise makes the approximation quality of the surrogate model worse compared to a noise-free case, the differences in observed function evaluations f is dominated by differences in $\mathbb{E}f$. Therefore, the surrogate model still points into the right direction.

In the right plot, the trust-region is in a flat area of the objective function. Therefore, the differences in observed function evaluations f are dominated by differences in the realized noise and do not reflect differences in $\mathbb{E}f$. As a result, the surrogate model points in the wrong direction.



(a) Noise with a steep function

(b) Noise with a flat function

Figure 8. Effect of noise on a surrogate model

This simple illustration shows that setting the sample size based on the variance of the error term and the trust-region radius alone is not sufficient to ensure that sample sizes are close to optimal. Since objective functions are typically steeper in the beginning and flatter as we get closer to the

optimum, the optimal sample size will typically be increasing in the iteration. However, it is very hard to pick an optimal schedule for this. Therefore, approaches that require the user to set a sequence of sample sizes as a function of the iteration counter require a lot of trial and error in practice. This motivates us to develop a fully adaptive approach to selecting the sample sizes in *tranquilo*.

5.2 Core ideas for noise handling

5.2.1 Determining m^{model}.

Our approach is based on the observation that the effects of noise on model quality are similar to the effects of approximation error and can, therefore, be handled in a similar way.

Approximation error is introduced by the fact that a quadratic surrogate model is not able to capture the shape of the objective function exactly. The tuning parameter to govern the size of the approximation error is the trust-region radius. A larger radius means a larger approximation error. Talyor-like error bounds ensure that by making the radius small enough, the approximation error can be made arbitrarily small. Of course, a small radius comes with the cost of smaller step sizes and slower progress. Therefore, it is not a goal to make the approximation error as small as possible but to make the radius as large as possible while ensuring that the model is good enough to produce suitable candidate points.

To achieve this balance, trust-region optimizers use ρ, the ratio of actual to predicted improvement, as a measure of model quality. If the actual improvement is larger or similar to the predicted improvement, the model was good enough to find suitable candidates, and, therefore, the radius can be increased or kept constant. Otherwise, the radius can be decreased.

Random error is introduced by the fact that observations of the objective function are noisy. The tuning parameter to govern the size of the random error is the number of repeated function evaluations. Under regularity conditions, a law of large numbers ensures that making m^{model} large enough will make the random error arbitrarily small. It is worth emphasizing that making the error small comes at the cost of more function evaluations and it is, therefore, not a goal to make the error as small as possible but to make it just small enough to ensure that the model is good enough to find suitable candidates.

The presence of random error does, of course, not alleviate the approximation error. If ρ is calculated as usual it reflects both kinds of errors:

$$\rho = \frac{f(x^*, \xi) - f(x^* + s, \xi)}{M^s(x^*) - M^s(x^* + s)} = \frac{\text{Actual Improvement}}{\text{Expected Improvement}}$$

To obtain a measure that mostly reflects approximation error, we could, of course, replace the noisy function evaluations with averages over multiple repetitions. However, this again requires

determining a sample size. We, therefore, first focus on finding a measure ρ^{noise} that only reflects the effect of random error on the model's ability to produce good candidate points.

We start by noting that the denominator of ρ is made up of quadratic models and that any alternative measure that puts a similar focus on the model's ability to produce good candidate points will likely have quadratic models in the denominator as well. If ρ^{noise} should not be affected by approximation error, we have to replace all occurrences of the objective function f in the numerator with a quadratic model that approximates f. Of course, the best quadratic approximation of f we have available is the surrogate model M^s.

We therefore construct ρ^{noise} with a simulation approach in which we use the current surrogate models M^v and M^s as a stand-in for the residual and objective functions. Using M^v we can create a sample of "true" residuals for each point in the current set of model points $\mathcal{X}_t^{model}$. Using an estimate of the noise variance, we can then simulate noisy function evaluations. On the simulated function evaluations, we can fit simulated vector models $M^{v,sim}$, aggregate them into simulated scalar models $M^{s,sim}$, and solve the simulated trust-region subproblem. This yields a candidate step s^{sim}. Given these ingredients, we can calculated ρ^{noise} as follows:

$$\rho^{noise} = \frac{M^s(x^*) - M^s(x^* + s^{sim})}{M^{s,sim}(x^*) - M^{s,sim}(x^* + s^{sim})} \tag{5.1}$$

All steps in the calculation of the simulated candidate step s^{sim} completely mirror the steps done to calculate the normal candidate step s in tranquilo. The only difference is that the true objective function f is replaced by the surrogate model M^s and that noisy function evaluations are replaced by their simulated counterparts. This means that ρ^{noise} is a pure measure of the effects of random error on the model's ability to produce good candidate points. As both the numerator and denominator are made up of quadratic models, it does not contain any approximation error.

In our practical implementation, we repeat the simulation b times to create a vector of ρ^{noise} values. This vector can then be used to adjust m_t^{model}.

5.2.2 Determining m^{accept}.

In the absence of noise, accepting or rejecting a candidate step s boils down to the simple question of whether $f(x_t^* + s)$ is smaller than $f(x_t^*)$. In the presence of noise, this turns into a question of expected values: Is $\mathbb{E}f(x_t^* + s, \xi)$ smaller than $\mathbb{E}f(x_t^*, \xi)$? This is a hypothesis test.

Empirical economists who collect data frequently have to make decisions about sample sizes. Collecting data is expensive, but collecting too little data might render non-zero effects statistically insignificant. The method of choice for determining sample sizes to alleviate this problem is power analysis.

For simplicity, we assume that our test statistic of interest –the difference in means between function evaluations at the current and candidate point– is normally distributed. This can either be seen as

a finite sample assumption or justified with asymptotic arguments. Given this assumption, we need to fix three ingredients for a power analysis: First, the significance level of the hypothesis test that is going to be performed. Second, the desired power for detecting an effect. Third, the minimal detectable effect size.

We treat the first two as tuning parameters of the algorithm and set them to maximize performance in benchmarks. For the minimal detectable effect size, this cannot be done because it depends on the scale of the objective function. However, the solution of the trust-region subproblem generates an expected improvement as a by-product. We use this expected improvement as a minimal detectable effect size.

The results of the power analysis, together with the existing number of function evaluations at x_t^* and $x_t^* + s$, can then be used to calculate optimal values for m_{t1}^{accept} and m_{t2}^{accept} that minimize the number of additionally required function evaluations while achieving the desired power. In our practical implementation we also keep the number of function evaluations used in the acceptance step between a lower and upper bound that can be set by the user. The details of this implementation are described in Subsection 5.3.5.

5.3 Adding noise handling to tranquilo

5.3.1 Noisy-trustregion-framework.

In this section, we describe the changes to the core algorithm framework as depicted in Algorithm 1 that are necessary to make *tranquilo* robust to noise. The modified algorithm is shown in Algorithm 3. The changes are highlighted in green.

The noisy version of *tranquilo* contains two additional inputs: m_0, which determines how often the objective function is evaluated at the start parameters, and m_0^{model}, which is the initial value for the adaptively chosen number of repeated function evaluations. m_0 needs to be larger or equal to two, such that we cannot just get an estimate of the function value at the start parameters but also an estimate of the noise variance. In our benchmarks, we set m_0 to 5 and m_0^{model} to 1.

The first thing that changes in the algorithm is that the History $\mathcal{H}_t$ is now initialized with a set of function evaluations at the start parameters instead of just one function evaluation. In general, each parameter vector in the history can now be associated with several observed function evaluations, and the number of function evaluations varies over time. As a consequence, $\mathcal{F}_t^{model}$ and $\mathcal{R}_t^{model}$ are now replaced by $\overline{\mathcal{F}}_t^{model}$ and $\overline{\mathcal{R}}_t^{model}$ which contain averages over multiple function or residual evaluations at each model point. Similarly, the initial vector model M_0^v is now initialized with the average of the function evaluations as intercept terms. All other coefficients stay at zero.

The main loop of *tranquilo* proceeds as before through the process of scanning the history, filtering existing points, sampling new points, fitting and aggregating vector models, and solving the trust-region subproblem. The two while loops for stagnation handling are also unchanged.

Algorithm 3: *Tranquilo* algorithm (noisy case)

Input: Starting point x_0^*, initial trust-region radius Δ_0^{region}, target sample size n^{target}, search factor γ^{search}, minimum step size s^{min}, sample increment n_{stag}^{drop}, maximum number of iterations t^{max}, maximum number of trials to avoid stagnation n_{stag}^{max}, lower and upper bounds l and u, the number of function evaluations at the start parameters m_0, and the initial value for the number of repeated function evaluations m_0^{model}.

Initialize history with $\mathcal{H}_0 = \{(x_0^*, \{r_j(x_0^*) : j = 1, ..., m_0\})\}$
Initialize vector model M_0^v with intercept terms at $\frac{1}{m_0}\sum_{j=1}^{m_0} r_j(x_0^*)$ and all other coefficients set to zero
for t=0,1,...,t^{max} **do**
    Calculate the search radius $\Delta_t^{search} = \gamma^{search}\Delta_t^{region}$
    Calculate the effective trust-region R_t based on x_t^*, Δ_t^{region}, l and u
    Scan the history for existing points $\mathcal{X}_t^{existing} = \{x \in \mathcal{H}_t : \|x_t^* - x\| \leq \Delta_t^{search}\}$
    Filter existing points: $\mathcal{X}_t^{filtered} = Filter(\mathcal{X}_t^{existing})$
    if $|\mathcal{X}_t^{filtered}| < n^{target}$ **then**
        Sample $n^{target} - |\mathcal{X}_t^{filtered}|$ new points in the trust-region: $\mathcal{X}_t^{new} = Sample(\mathcal{X}_t^{filtered}, R_t, n^{target})$
        $\mathcal{X}_t^{model} = \mathcal{X}_t^{filtered} \cup \mathcal{X}_t^{new}$
    else
        $\mathcal{X}_t^{model} = \mathcal{X}_t^{filtered}$
    end
    Build a vector model $M_t^v = Fit(\mathcal{X}_t^{model}, \overline{\mathcal{R}}_t^{model}, M_{t-1}^v, R_t)$
    Aggregate the vector model: $M_t^s = Aggregate(M_t^v)$
    Solve the surrogate problem: $s_t = Subsolve(M_t^s, R_t)$
    while $|\mathcal{X}_t^{model}| > n^{target}$ *and* $\|s_t\| \leq s^{min}$ **do**
        Reduce the sample: $\mathcal{X}_t^{reduced} = Drop(\mathcal{X}_t^{model}, n_{stage}^{drop}, \Delta_t^{region})$ and set $\mathcal{X}_t^{model} = \mathcal{X}_t^{reduced}$
        Build a vector model $M_t^v = Fit(\mathcal{X}_t^{model}, \overline{\mathcal{R}}_t^{model}, M_{t-1}^v, R_t)$
        Aggregate the vector model: $M_t^s = Aggregate(M_t^v)$
        Solve the surrogate problem: $s_t = Subsolve(M_t^s, R_t)$
    end
    $n_{stag} = 0$
    while $\|s_t\| \leq s^{min}$ *and* $n_{stag} \leq n_{stag}^{max}$ **do**
        Reduce the sample: $\mathcal{X}_t^{reduced} = Drop(\mathcal{X}_t^{model}, n_{stag}^{drop}, \Delta_t^{region})$
        Sample new points in the trust-region: $\mathcal{X}_t^{new} = Sample(\mathcal{X}_t^{reduced}, R_t, n^{target})$ and set $\mathcal{X}_t^{model} = \mathcal{X}_t^{reduced} \cup \mathcal{X}_t^{new}$
        Build a vector model $M_t^v = Fit(\mathcal{X}_t^{model}, \overline{\mathcal{R}}_t^{model}, M_{t-1}^v, R_t)$
        Aggregate the vector model: $M_t^s = Aggregate(M_t^v)$
        Solve the surrogate problem: $s_t = Subsolve(M_t^s, R_t)$
        $n_{stag} = n_{stag} + 1$
    end
    Estimate the noise variance of the objective and residual functions $\sigma_t, \Sigma_t = Varest(\mathcal{H}_t, R_t)$
    Calculate $\Delta M_t^s = M_t^s(x_t^*) - M_t^s(x_t^* + s_t)$
    Accept or reject the step and calculate a measure of progress $(x_{t+1}^*, \rho_t) = Accept(x_t^*, s_t, \Delta M_t^s, \sigma_t)$
    Simulate the effect of noise on ρ: $\rho^{noise} = \{\rho_1^{noise}, \dots \rho_b^{noise}\} = SimNoise(\mathcal{X}^{model}, \mathcal{R}^{model}, M_{t-1}^v, M_t^v, R_t, \Sigma_t)$
    Adjust the number of repeated function evaluations: $m_{t+1}^{model} = AdjustRep(\rho^{noise}, \rho_t, m_t^{model})$
    Adjust the trust-region radius: $\Delta_{t+1}^{region} = AdjustRadius(\Delta_t^{region}, \rho_t, s_t, m_t^{model}, m_{t+1}^{model})$
    if $x_{t+1}^* \neq x_t^*$ *and* $Converged(\mathcal{H}_t, M_t^s, x_t^*, x_{t+1}^*)$ **then**
        break
    end
end

The major changes appear when the original algorithm would have proceeded with the acceptance step. In the noisy case, we first estimate the variance of the noise term in the objective function (σ_t) as well as the variance-covariance matrix of the noise terms in the residual function (Σ_t). The actual implementation of the noise estimation is again a replaceable component, which is further described in Subsection 5.3.2.

While the expected improvement is calculated as before, the acceptance step now takes the estimated noise variance σ_t into account. Implementations of noise robust acceptance steps are described in Subsection 5.3.5.

After the acceptance step, two new steps are introduced. The first is to simulate our noisy measure of model quality ρ^{noise}, and the second is to adjust the number of repeated function evaluations m_t^{model} based on the simulated values. Both are replaceable components, which are further described in Subsection 5.3.3 and Subsection 5.3.4. Finally, the trust-region radius is adjusted as before. The only difference is that it now takes two additional arguments m_t^{model} and m_{t+1}^{model}. This can be used to skip radius decreases in situations where the number of repetitions was increased.

5.3.2 Estimation of noise variance.

The literature on noisy optimization generally distinguishes between two types of noise: Additive noise is a noise term with fixed variance over the entire parameter space that is added to the objective function. Multiplicative noise is a noise term that enters the objective function as a multiplicative factor, and therefore, the effective variance of the noise varies with the value of the objective function. In least-squares optimization, the noise term is added to the residual function, and therefore, even additive noise leads to a noise term whose variance varies over the parameter space.

In *tranquilo*, we treat the noise term as constant over the current trust-region. This can be seen as a locally constant approximation to more general noise terms. We do not make any assumptions about how the noise term varies between trust-regions. We distinguish between Σ_t, the variance-covariance matrix of the noise terms in the residual function, and σ_t, the variance of the noise term in the objective function.

While we implement the noise estimation as a replaceable component, we provide just one implementation: We first calculate a search radius $\Delta_t^{varest} = \gamma^{varest} \Delta_t^{region}$ and scan the history for function evaluations within this radius of the current point. Out of these points, we only keep the ones at which the objective function was evaluated at least m^{varest} times. Next, we de-mean the function and residual evaluations at each point. Finally, we calculate σ_t as the variance of the de-meaned function evaluations and Σ_t as the variance-covariance matrix of the de-meaned residual evaluations.

To make sure that at least one point exists in the current trust-region at which the function has been evaluated often enough to get a variance estimate, the minimal number of repeated function evaluations in the acceptance step m_{min}^{accept} needs to be set larger or equal to m_{min}^{varest}. In our benchmarks, we set $m_{min}^{varest} = 3$ and $m_{min}^{accept} = 4$.

5.3.3 Simulating ρ^{noise}.

The goal of this step is to simulate multiple instances of measures of model quality $\rho^{noise} = \{\rho_1^{noise}, \ldots, \rho_b^{noise}\}$ that can be used to adjust m_t^{model}. In principle, there are many possibilities for doing this, which can range from heuristics to computationally costly simulation approaches. Currently, we implement just one approach, which is based on simulations.

The approach for generating ρ^{noise} is described in Algorithm 4. The inputs of the algorithm are the current set of model points $\mathcal{X}_t^{model}$, the current and previous vector models M_{t-1}^v and M_t^v, the current effective trust-region R_t, the estimated variance-covariance matrix of the noise terms in the residual function Σ_t, the current parameter vector x_t^*, the number of simulations b, and the number of repeated function evaluations m_t^{model}. The first few inputs provide almost all ingredients for a standard fitting step in *tranquilo*. The only thing that is missing are the residuals at the model points $\mathcal{R}_t^{model}$, because those will be replaced by simulated counterparts.

The simulation starts by calculating the "true" and noise-free residuals at the model points. They are denoted by $\mathcal{R}_{sim,true}^{model}$ and calculated by evaluating the current vector model M_t^v at the model points. These "true" residuals play the role of the unobservable $\mathbb{E}r(x, \xi)$ during the simulation.

For each $\ell = 1, \ldots, b$ simulation draw, we start by creating averages of simulated noisy residuals, denoted by $\overline{\mathcal{R}}_{sim,\ell}^{model}$. These play the role of the average observed residuals $\overline{\mathcal{R}}^{model}$ in the simulation, i.e., they will be used to fit vector models $M_\ell^{v,sim}$. To capture residualized model fitting, the previous vector model M_{t-1}^v is used inside each simulated fitting step. The simulated vector models are then aggregated into simulated scalar models $M_\ell^{s,sim}$ and used to solve the simulated trust-region subproblem. This yields a candidate step s_ℓ^{sim}.

Given these ingredients, we can calculate the simulated measure of model quality ρ_ℓ^{noise} as in Equation 5.3.3. Since the simulated ρ_ℓ^{noise} mimics all steps of the actual algorithm, it is a pure measure of the effect of random error on the model's ability to produce good candidate points. All other errors, such as approximation error, as well as imperfect solutions of the trust-region subproblem or numerical imprecisions in the fitting process are reflected in the standard ρ but not in ρ^{noise}.

In our practical implementation, we set $b = 100$. This means that simulating ρ^{noise} incurs the computational overhead of fitting, aggregating and minimizing 100 surrogate models. While this overhead is much larger than an iteration of a typical trust-region algorithm, it is justified in a setting with an expensive objective function.

Algorithm 4: Simulating ρ^{noise}

Input: Model points $\mathcal{X}_t^{model}$, current and previous vector models M_{t-1}^v and M_t^v, the current effective trust-region R_t, The variance-covariance matrix of the noise terms in the residual function Σ_t, the current parameter vector x_t^*, the number of simulations b and the number of repeated function evaluations m_t^{model}.

Calculate "true" residuals $\mathcal{R}_{sim,true}^{model} = \{M_t^v(x) : x \in \mathcal{X}_t^{model}\}$

for $\ell = 1, \ldots, b$ **do**

  Simulate average noisy residuals $\overline{\mathcal{R}}_{sim,\ell}^{model}$ over m_t^{model} simulated noisy residuals that are created by adding noise draws from $N(0, \Sigma_t)$ to "true" residuals in $\mathcal{R}_{sim,true}^{model}$

  Fit a simulated vector model: $M_\ell^{v,sim} = Fit(\mathcal{X}_t^{model}, \overline{\mathcal{R}}_{sim,\ell}^{model}, M_{t-1}^v, R_t)$

  Aggregate the simulated vector model: $M_\ell^{s,sim} = Aggregate(M_\ell^{v,sim})$

  Solve the simulated trust-region subproblem: $s_\ell^{sim} = Subsolve(M_\ell^{s,sim}, R_t)$

  Calculate the simulated measure of model quality: $\rho_\ell^{noise} = \frac{M_t^s(x_t^*) - M_t^s(x_t^* + s_\ell^{sim})}{M_\ell^{s,sim}(x_t^*) - M_\ell^{s,sim}(x_t^* + s_\ell^{sim})}$

end

5.3.4 Adjusting m^{model}.

The goal of this step is to adjust the number of repeated function evaluations m_t^{model} based on the simulated ρ_ℓ^{noise} values. A simple possibility would be to calculate the average of the simulated ρ_ℓ^{noise} values and then adjust m_t^{model} in a very similar way to the adjustment of the trust-region radius. A drawback of this approach is that the denominator of the simulated ρ_ℓ^{noise} can be very small and therefore, a non-robust statistic like the average is strongly affected by a few outliers.

To avoid this problem, we choose an approach that is not based on the average but on the share of simulated ρ_ℓ^{noise} values below and above certain cutoffs. In particular, we use the following approach: ρ_{high}^{noise} and ρ_{low}^{noise} are cutoffs thate determine whether a simulated ρ_ℓ^{noise} is considered high or low. If more than π^{high} of the simulated ρ_ℓ^{noise} are high, we conclude that m_t^{model} is unnecessarily large and decrease it by one in order to save costly function evaluations in the next iteration. If this is not the case but more than π^{low} of the simulated ρ_ℓ^{noise} are high or the overall ρ_t is larger than ρ_{keep}, we conclude that m_t^{model} is just right and leave it unchanged. Otherwise, we increase m_t^{model} by one.

To improve robustness, we also keep the number of repeated function evaluations between a lower and upper bound. m_{min}^{model} is the minimal number of function evaluations, which we set to 1 in our benchmarks. m_{max}^{model} is the maximal number of function evaluations, which we set to 30.

5.3.5 Noisy acceptance steps.

The noisy acceptance step requires an additional input σ_t, over the noise-free acceptance step (Equation 3.9). It is a replaceable component, but we only provide one implementation based on the power analysis ideas described in Subsection 5.2.2.

$$(x^*_{t+1}, \rho_t) \;=\; Accept(x^*_t, s_t, \Delta M^s_t, \sigma_t)$$

As in the noise-free case, x^*_{t+1} is the candidate point for the next iteration, ρ_t is a measure of progress or model quality, ΔM^s_t is the expected improvement from taking step s_t, and now additionally, σ_t is the estimated variance of the noise term in the objective function.

In the noise-free case, we would, generally, accept the candidate step s_t if it yields any improvement over the current point. That is, if $f(x^*_t + s_t) < f(x^*_t)$. In the noisy case, we are ultimately interested in minimizing the $\mathbb{E}f$, and thus we would like to make the comparison $\mathbb{E}f(x^*_t + s_t, \xi) < \mathbb{E}f(x^*_t, \xi)$. However, this is, of course, not observed. Instead, we can try to reduce the effect of the noise by averaging multiple function evaluations at each point. Define the averages of the objective function at the candidate and current point as:

$$\bar{f}(x^*_t + s_t) \;=\; \frac{1}{m^{accept}_{t2}} \sum_{i=1}^{m^{accept}_{t2}} f(x^*_t + s_t, \xi_i) \quad \text{and} \quad \bar{f}(x^*_t) \;=\; \frac{1}{m^{accept}_{t1}} \sum_{j=1}^{m^{accept}_{t1}} f(x^*_t, \xi_j)$$

The noisy acceptance decision is then based on the following condition:

$$\text{Accept } s_t \iff \bar{f}(x^*_t + s_t) \;<\; \bar{f}(x^*_t) \tag{5.2}$$

Similarly to the noise-free case in Equation 2.1, we can compute ρ_t by replacing f with $\bar{f}$:

$$\rho_t \;=\; \frac{\bar{f}(x^*_t) - \bar{f}(x^*_t + s_t)}{M^s_t(x^*_t) - M^s_t(x^*_t + s_t)}$$

While the mechanics of the noisy acceptance step are straightforward, as alluded to in the previous sections, the difficulty stems from determining m^{accept}_{t1} and m^{accept}_{t2} such that the decision based on the averages $\bar{f}$ is a good proxy for the decision based on expected values $\mathbb{E}f$.

One way to solve this problem is to use a two-sample power analysis. For this we need to assume that

(1) $f(x^*_t + s_t, \xi_i)$ is independent of $f(x^*_t, \xi_j)$ for all i, j

(2) $\frac{1}{m^{accept}_{t1}} \sum_{j=1}^{m^{accept}_{t1}} f(x^*_t, \xi_j) \approx N\left(\mathbb{E}f(x^*_t), \sigma^2 / m^{accept}_{t1}\right)$

(3) $\frac{1}{m^{accept}_{t2}} \sum_{i=1}^{m^{accept}_{t2}} f(x^*_t + s_t, \xi_i) \approx N\left(\mathbb{E}f(x^*_t + s_t), \sigma^2 / m^{accept}_{t2}\right)$

(4) σ_t is a reasonable estimate of σ

Given a significance level $\alpha \in (0,1)$, a power level $1-\beta \in (0,1)$, and a minimal detectable effect size $M_t^s(x_t^*) - M_t^s(x_t^* + s_t)$, by choosing m_{t1}^{accept} and m_{t2}^{accept} under following condition, we can guarantee that the noisy acceptance condition (Equation 5.2) is done with a significance level of α and a power level of $1-\beta$:

$$\frac{m_{t1}^{accept} m_{t2}^{accept}}{m_{t1}^{accept} + m_{t2}^{accept}} \geq \left[\frac{\Phi^{-1}(1-\alpha) + \Phi^{-1}(1-\beta)}{(M_t^s(x_t^*) - M_t^s(x_t^* + s_t))/\sigma_t} \right]^2 \tag{5.3}$$

Since we still want to minimize the number of function evaluations, the actual choice of m_{t1}^{accept} and m_{t2}^{accept} is based on the following problem:

$$\underset{m_{t1}^{accept}, m_{t2}^{accept} \in \mathbb{N}}{\text{minimize}} \; m_{t1}^{accept} + m_{t2}^{accept} \text{ s.t. Equation 5.3 holds}$$

A detailed derivation of Equation 5.3 is provided in Appendix C.

5.3.6 Noisy radius adjustment.

The noise robust radius adjustment is identical to the noise-free version, except that radius decreases are skipped if m_{t+1}^{model} is larger than m_t^{model}. This successfully prevents the trust-region radius from collapsing to zero before a suitable value for m^{model} is found.

5.4 Benchmarking

To test the performance of *tranquilo* in a noisy setting, we use the bootstrapped version of the Moré-Wild benchmark set. Following Cartis, Fiala, et al. (2019), we add identical and independently normal-distributed noise terms to each residual. We choose a large standard deviation of 1.2 to create a challenging benchmark set (compared to 0.01 in Cartis, Fiala, et al. (2019)). Note that the scale of the residuals in the Moré-Wild benchmark varies drastically across problems. This means that even though we add the same amount of noise to each residual, we obtain problems with very different difficulties and with very different optimal sequences of sample sizes.

Since *tranquilo* is fully adaptive, we only run it in one configuration. For *DFO-LS*, we choose configurations with three different sample sizes. Note that in *DFO-LS*, $m^{model} = m^{accept}$. Since it is very hard to pick optimally increasing sequences of sample sizes in practice, we restrict ourselves to fixed sequences of 3, 5, and 10 function evaluations at each point.

Since we are interested in minimizing the expected value of our objective functions, the convergence test is based on evaluating the noise-free objective function at the parameter vectors generated by the algorithm. Then the convergence test is as before, but the tolerance level τ is relaxed to 0.1 to reflect that we cannot expect the same precision for noisy and noise-free optimization problems. The results of the benchmark are shown in Figure 9.

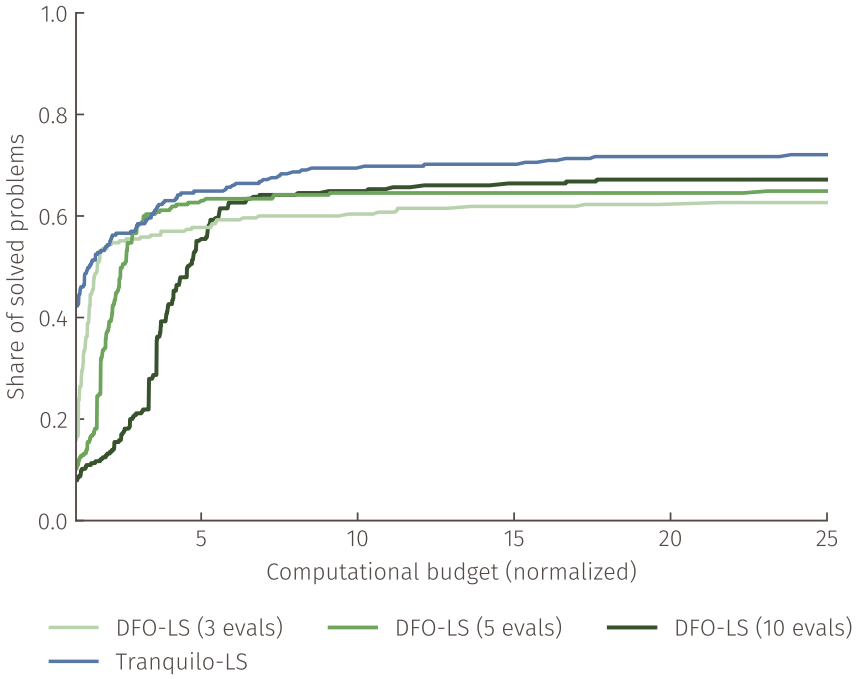


Figure 9. Comparison of least-squares optimizers on an augmented Moré-Wild benchmark set with added noise. The noise is normally distributed with a standard deviation of 1.2. The *x*-axis shows the normalized computational budget. The computational budget is measured in terms of objective function evaluations needed by the optimizers. Normalized means that the number of function evaluations each algorithm needed to solve a given problem is divided by the number of function evaluations the fastest algorithm needed to solve that problem. The different *DFO-LS* configurations vary in the number of repeated function evaluations at each point. *tranquilo* is fully adaptive and therefore does not need multiple configurations. The plot shows that *tranquilo* outperforms the *DFO-LS* configurations in speed and robustness.

We see that *DFO-LS* with three evaluations solves some problems very quickly but then stagnates abruptly. Using 5 evaluations at each point makes the algorithm slower but helps to solve more problems. The pattern repeats for 10 evaluations, even though only a few additional problems are solved by switching from 5 to 10 evaluations. This shows that it is very hard to pick a sample size that works well for several problems, and in fact, the sample sizes 3, 5, and 10 are already the result of some trial and error in which the whole benchmark set was solved multiple times.

Tranquilo starts at a low sample size and can, therefore, solve easy problems very quickly. If necessary, the sample size is increased, and therefore, *tranquilo* solves more problems than any configuration of *DFO-LS*. In total, *tranquilo* is the fastest algorithm for more than 40 % of the problems. Moreover, its performance-profile is consistently above the performance profiles of all *DFO-LS* configurations.

6 conclusion

This paper presents the *tranquilo* algorithm, an optimizer for noisy nonlinear least-squares problems with expensive objective functions. A typical situation in which such problems arise is the estimation of econometric models using the method of simulated moments (MSM). *Tranquilo* improves over existing least-squares optimizers in two important ways: By introducing a line-search and speculative sampling approach, the algorithm becomes more parallelizable and the solution can be accelerated if multi-core machines are available. By introducing novel approaches for adaptive noise handling, the algorithm can solve noisy optimization problems without requiring the user to set any advanced algorithm parameters.

We show that in a noise-free and serial setting, tranquilo is roughly competitive with other state-of-the-art optimizers. The parallel version of *tranquilo* is much faster that the serial version. For noisy objective functions, tranquilo outperforms existing optimizers.

Appendix A Notation

Table A.1. Algorithm constants

Symbol	Description
$p \in \mathbb{N}$	Number of parameters in the optimization problem
$k \in \mathbb{N}$	Number of least-squares residuals. 1 for scalar problems
$n^{target} \in \mathbb{N}$	Target for the number of points used to construct surrogate models. Independent of the number of evaluations at each point in the noisy case. Usually $p + 1$ for least-squares optimizers
$n^{filter} \in \mathbb{N}$	Maximum number of points that remain after filtering. Typically larger than n^{target}
$l \in \mathbb{R}^p \cup -\infty$	Lower bounds for parameters
$u \in \mathbb{R}^p \cup \infty$	Upper bounds for parameters
$\gamma^{search} \in \mathbb{R}^+$	Search radius factor, usually ≥ 1
$s^{min} \in \mathbb{R}^+$	Minimum step size
$n^{max}_{stag} \in \mathbb{N}$	Maximum number of trials to avoid stagnation
$n^{drop}_{stag} \in \mathbb{N}$	Sample increment
$t^{max} \in \mathbb{N}$	Maximum number of iterations
$d^s \in \mathbb{N}$	Number of free coefficients of a scalar surrogate model
$d^v \in \mathbb{N}$	Number of free coefficients of each individual model in a vector surrogate model
$n^{cores} \in \mathbb{N}$	Number of available cores
$n^{batch} \in \mathbb{N}$	Batch size

Table A.2. Component specific constants

Symbol	Description
AdjustRadius	
$\rho^{inc} \in \mathbb{R}^+$	Radius shrinking cutoff
$\rho^{dec} \in \mathbb{R}^+$	Radius expansion cutoff
$\gamma^{inc} \in \mathbb{R}^+$	Radius expansion factor
$\gamma^{dec} \in \mathbb{R}^+$	Radius shrinking factor
$\Delta^{max} \in \mathbb{R}^+$	Radius bound
$c^{ls} \in \mathbb{R}^+$	Large radius cut-off
Converge	
$\varepsilon^{fatol} \in \mathbb{R}^+$	Convergence absolute tolerance objective function
$\varepsilon^{frtol} \in \mathbb{R}^+$	Convergence relative tolerance objective function
$\varepsilon^{gatol} \in \mathbb{R}^+$	Convergence absolute tolerance surrogate model gradient
$\varepsilon^{grtol} \in \mathbb{R}^+$	Convergence relative tolerance surrogate model gradient
$\varepsilon^{xatol} \in \mathbb{R}^+$	Convergence absolute tolerance parameters
$\varepsilon^{xrtol} \in \mathbb{R}^+$	Convergence relative tolerance parameters
Varest	
γ^{varest}	Factor to calculate a search radius for points used for noise variance estimation
m_{min}^{varest}	Minimal number of function evaluations required to use a point for variance estimation
Accept	
m_{min}^{accept}	Minimal number of repeated function evaluations for acceptance steps
m_{max}^{accept}	Maximal number of repeated function evaluations for acceptance steps
SimulateNoise	
b	Number of simulation runs for the calculation of ρ^{noise}
AdjustRep	
m_0	Number of repeated function evaluations at start parameters
ρ_{high}^{noise}	Threhshold for a simulated ρ^{noise} to be considered high
ρ_{low}^{noise}	Threhshold for a simulated ρ^{noise} to be considered low
π^{high}	Minimal share of high ρ^{noise}-estimates required to decrease m_t^{model}
π^{low}	Minimal share of low ρ^{noise}-estimates required to increase m_t^{model}
ρ_{keep}	Threshold for ρ_t to be considered good enough that m does not have to be increased. This refers to the overall ρ, not ρ^{noise}
m_{min}^{model}	Minimal number of repeated function evaluations for surrogate model construction
m_{max}^{model}	Maximal number of repeated function evaluations for surrogate model construction

Table A.3. Internal algorithm variables

Symbol	Description
$x_t^* \in \mathbb{R}^p$	Accepted parameter vector at the beginning of iteration t. Also serves as trustregion center
Δ_t^{region}	Trust region radius in iteration t
Δ_t^{search}	Search radius, defined as $\gamma^{search}\Delta_t^{region}$
R_t	Effective trustregion in iteration t. If no bounds are binding, R_t is defined as a ball with center x_t^* and radius Δ_t^{region}. If bounds are binding, R_t is defined as a hypercube with the same volume as a ball with radius Δ_t^{region} that contains x_t^* and respects bound constraints
$\mathcal{H}_t$	History of function evaluations and parameter vectors up to period t
$\mathcal{X}_t^{existing} \subset \mathbb{R}^p$	Points inside the search radius for which the function has previously been evaluated
$\mathcal{X}_t^{filtered} \subset \mathbb{R}^p$	Filtered existing points
$\mathcal{X}_t^{new}$	Newly sampled points in iteration t. Defined as $\mathcal{X}^{filtered} \cup \mathcal{X}_t^{new}$
$\mathcal{X}_t^{model}$	Model points
$\mathcal{R}_t^{existing}, \mathcal{R}_t^{filtered}, \ldots$	Least-squares residuals evaluated on the corresponding set of points
$\mathcal{F}_t^{existing}, \mathcal{F}_t^{filtered}, \ldots$	Objective function evaluations on the corresponding set of points
$M_t^s \in \mathscr{M} = \mathbb{R} \times \mathbb{R}^p \times \mathbb{R}^{p \times p}$	Scalar quadratic model defined by an intercept, gradient-terms and hessian-terms
$M_t^v \in \mathscr{M}^k$	Vector model consisting of one scalar model per least-squares residual
$\Delta f_t \equiv f(x_t^*) - f(x_t^* + s_t)$	Actual improvement through step s_t
$\Delta M_t^s \equiv M_t^s(x_t^*) - M_t^s(x_t^* + s_t)$	Expected improvement through step s_t
$\overline{\mathcal{R}}_t^{existing}, \overline{\mathcal{R}}_t^{filtered}, \ldots$	Averaged Least-squares residuals evaluated on the corresponding set of points
$\overline{\mathcal{F}}_t^{existing}, \overline{\mathcal{F}}_t^{filtered}, \ldots$	Averaged function evaluations on the corresponding set of points
σ_t	Estimate of the noise variance in the objective function in iteration t
Σ_t	Estimate of the noise variance-covariance matrix in the residual function in iteration t
m_t^{model}	Number of repeated function evaluations for surrogate model construction in iteration t
m_{t1}^{accept} and m_{t2}^{accept}	Number of repeated function evaluations at x_t^* and $x_t^* + s_t$ for the acceptance step in iteration t. These are actually component variables of acceptance steps but listed here because all noise-robust acceptance steps use these variables

Table A.4. Component functions

Symbol	Description
$\mathscr{X}_t^{filtered} = Filter(\mathscr{X}_t^{existing})$	Filter applied to sample of existing points
$\mathscr{X}_t^{new} = Sample(\mathscr{X}_t^{filtered}, R_t, n^{target})$	Sample new points inside the trustregion
$M_t^v = Fit(\mathscr{X}_t^{model}, \mathscr{R}_t^{model}, M_{t-1}^v, R_t)$	Fit a quadratic model scaled to the trustregion
$M_t^s = Aggregate(M_t^v)$	Aggregate vector model into scalar model
$s_t = Subsolve(M_t^s, R_t)$	Solve the trustregion subproblem
$\mathscr{X}_t^{reduced} = Drop(\mathscr{X}_t^{model}, n^{drop}, \Delta_t^{region})$	Drop the n^{drop} worst points
$(x_{t+1}^*, \rho_t) = Accept(x_t^*, s_t, \Delta M_t^s, \sigma_t)$	Accept or reject the proposed step and calculate a measure of progress. The argument σ_t is only used in the noisy case
$\Delta_{t+1}^{region} = AdjustRadius(\Delta_t^{region}, \rho_t, s_t)$	Adjust the trustregion radius
$Converged(\mathscr{H}_t, M_t^s, x_t^*, x_{t+1}^*)$	Check for convergence
$\sigma_t, \Sigma_t = Varest(\mathscr{H}_t, R_t)$	Estimate the noise variance
$\rho^{noise} = SimNoise(\mathscr{X}^{model}, \mathscr{R}^{model}, M_{t-1}^v, M_t^v, R_t, \Sigma_t)$	Simulate ρ that would obtain in the absence of Approximation error due to noise. $\rho^{noise} = (\rho_1^{noise}, \ldots, \rho_b^{noise})$ is a vector of simulated ρ's.
$m_{t+1}^{model} = AdjustRep(\rho^{noise}, \rho_t, m_t^{model})$	Adjust the number of repeated function evaluations for surrogate model construction

Table A.5. Mathematical symbols

Symbol	Description
$\|\|x\|\|$	The Euclidean norm of a vector x
$N(\mu, \Sigma)$	A (multivariate) Normal distribution with mean μ and variance-covariance (matrix) Σ
$\Phi(x)$	The cumulative distribution function of the standard Normal distribution evaluated at x
$\lceil x \rceil$	The smallest integer greater than or equal to x

Appendix B Subsolvers

B.1 GQTPAR

In the SP-Ball case, *GQTPAR* finds an exact solution to the trust-region subproblem (Moré and Sorensen (1983)), which satisfies

$$(H + \lambda I)\ s^* \ = \ -g \tag{B.1}$$

where g is the model gradient, H denotes the model Hessian, and I is the identity matrix. *GQTPAR* determines the Lagrange multiplier $\lambda \geq 0$ such that the matrix $(H + \lambda I)$ is positive definite and $\lambda(1 - \|s^*\|) = 0$. The latter is a complementary slackness condition which states that at least one of the quantities λ and $(1 - \|s^*\|)$ must be zero at the optimum s^*. Recall that in the problem SP-Ball, the subspace B is a ball with a center of 0 and a radius of 1, defined as $B := \{s \in \mathbb{R}^p : \|s\| \leq 1\}$. When the solution s^* is interior to B, i.e. $\|s^*\| < 1$, then $\lambda = 0$ and *GQTPAR* terminates immediately. Otherwise, when s^* lies on the boundary of B, i.e. $\|s^*\| = 1$, $\lambda > 0$, and Newton's method is applied to find the value of λ such that $\|s^*\| = 1$ is satisfied. Rearranging Equation B.1, Moré and Sorensen (1983) show that the unique solution s^*, which depends on λ, is defined as

$$s^*(\lambda) \ = \ -(H + \lambda I)^{-1}\ g \tag{B.2}$$

for $\lambda > 0$ sufficiently large so that $(H + \lambda I)$ is positive definite and $\|s(\lambda)\| = 1$. To obtain s^*, one first needs an expression for λ. Starting with an initial guess, *GQTPAR* updates λ in each iteration ℓ of Algorithm 5 via

$$\lambda^{(\ell+1)} \ = \ \lambda^{(\ell)} - \frac{\phi(\lambda^{(\ell)})}{\phi'(\lambda^{(\ell)})} \tag{B.3}$$

where the function $\phi(\lambda^{(\ell)})$ is defined as:

$$\phi(\lambda^{(\ell)}) \ = \ \frac{1}{\|s(\lambda^{(\ell)})\| - 1} \tag{B.4}$$

and $\phi'(\lambda^{(\ell)})$ is the first derivative of $\phi(\lambda^{(\ell)})$ with respect to $\lambda^{(\ell)}$. *GQTPAR* finds the optimal $\lambda^{(\ell)}$ by applying Newton's root finding method to the function $\phi(\lambda^{(\ell)})$ in Equation B.4, which is almost

linear around the optimal $\lambda^{(\ell)}$ (Nocedal and Wright (2006)). Before updating $\lambda^{(\ell)}$, however, an expression for $s(\lambda^{(\ell)})$ satisfying Equation B.2 is needed. *GQTPAR* obtains a candidate $s(\lambda^{(\ell)})$ by factorizing the model Hessian H via Cholesky factorization and solving the resulting linear system (see lines 4 and 5 of Algorithm 5). Conn, Gould, and Toint (2000) show that Equation B.3 can be simplified to the updating formula in line 11 of Algorithm 5, which makes the dependence on s_ℓ apparent. The vector q_ℓ is the solution to the linear system in line 10 of Algorithm 5, where R denotes the upper triangular matrix of H. The details are omitted here for brevity but are available in Conn, Gould, and Toint (2000). With expressions for s_ℓ and q_ℓ in hand, *GQTPAR* calculates their respective norms $\|s_\ell\|$ and $\|q_\ell\|$, and updates $\lambda^{(\ell+1)}$ in line 11 of Algorithm 5.

Note that in line 2 of Algorithm 5, $\lambda^{(\ell)}$ is safeguarded. This is necessary to ensure that the matrix $(H + \lambda^{(\ell)})I$ is positive definite and its Cholesky factorization exists. For details on the safeguarding procedure, see Moré and Sorensen (1983) or Nocedal and Wright (2006). This concludes Algorithm 5 for the "easy case".

Algorithm 5: GQTPAR algorithm - The "easy case"

Input: Initial guess s_0, $\lambda^{(0)}$, $\lambda_L^{(0)}$, $\lambda_U^{(0)}$
1 **for** ℓ=0,1,2,... **do**
2 Safeguard $\lambda^{(l)}$ to obtain $\lambda_S^{(\ell)}$
3 **if** $H + \lambda^{(\ell)}I$ *is positive definite* **then**
4 Factor $H + \lambda^{(l)}I = R^T R$
5 Solve $s_\ell = -(R^T R)^{-1} g$
6 **end**
7 Update $\lambda_L^{(\ell)}$, $\lambda_U^{(\ell)}$, $\lambda_S^{(\ell)}$
8 Check convergence criteria
9 **if** $H + \lambda^{(l)}I$ *is positive definite and* $g \neq 0$ **then**
10 Solve $q_\ell = (R^T)^{-1} s_\ell$
11 Set $\lambda^{(\ell+1)} = \lambda^{(\ell)} + \left(\frac{\|s_\ell\|}{\|q_\ell\|}\right)^2 \left(\|s_\ell\| - 1\right)$
12 **else**
13 Set $\lambda^{(\ell+1)} = \lambda_S^{(\ell)}$
14 **end**

There may be situations, where $(H + \lambda^{(\ell)}I)$ is positive definite but the solution s^* to Equation B.1 is not unique. This is what Moré and Sorensen (1983) call the "hard case", which is not described in Algorithm 5. We refer the interested reader to Moré and Sorensen (1983) and Conn, Gould, and Toint (2000) for details. In short, in the "hard case", Equation B.1 is replaced by

$$\left(H - \lambda_1 I\right)(s^* + \tau z) \;=\; -g \tag{B.5}$$

where z is the eigenvector of the model Hessian H corresponding to the first eigenvalue λ_1 of H. Moreover, z is such that $\|s + \tau z\ (\lambda)\| = 1$ for some τ.

B.2 BNTR

BNTR stands for "Bounded Newton Trust-Region" algorithm and was developed for the Toolkit of Advanced Optimization (Dener et al. (2021)). It employs a trust-region-like approach combined with an active set method to solve the bound-constrained problem SP-Cube. A search direction of the candidate step is considered "active" if it lies at the boundary of the trust-region. The active and inactive sets are defined as follows (Bertsekas (1982))

$$\begin{aligned}
\text{lower bounded: } \mathscr{L}(s) &= \{i : s_i \leq l_i \wedge g(s)_i > 0\}, \\
\text{upper bounded: } \mathscr{U}(s) &= \{i : s_i \geq u_i \wedge g(s)_i < 0\}, \\
\text{fixed: } \mathscr{F}(s) &= \{i : l_i = u_i\}, \\
\text{active-set: } \mathscr{A}(s) &= \mathscr{L}(s) \cup \mathscr{U}(s) \cup \mathscr{F}(s), \\
\text{inactive-set: } \mathscr{I}(s) &= \{1, 2, \ldots, n\} \setminus \mathscr{A}(s).
\end{aligned}$$

where l_i and u_i are the lower and upper bound on the ith search direction in s, respectively. Instead of fitting a full surrogate model, *BNTR* uses a simplified quadratic model in the surrogate step in line 5 of Algorithm 6. In particular, it solves for the minimizer r_ℓ of the reduced quadratic model $M_\ell^r(r)$ for the unconstrained (i.e. inactive) search directions only via a conjugate gradient method. The available methods are *Conjugate Gradient*, *Steihaug-Toint*, and *TRSBOX*. The reduced model is formed using the reduced model gradient g^r and the reduced model Hessian H^r based on the inactive set $\mathscr{I}(s)$, i.e. the unbounded search directions.

With the reduced conjugate gradient step r_ℓ in hand, *BNTR* constructs a new candidate step p_ℓ (line 6 of Algorithm 6). It does so by projecting r_ℓ onto the lower and upper bounds of the active set $\mathscr{A}(s)$

$$p = \begin{cases} l_i & \text{if } s_i < l_i \\ u_i & \text{if } s_i > u_i \\ r_i & \text{otherwise} \end{cases} \qquad \text{(bound-projection)}$$

where the subscript ℓ is omitted for readability. Similar to other trust-region optimizers, acceptance of the candidate step p_ℓ is determined based on the ratio of the actual over expected improvement of the surrogate model (see line 7 of Algorithm 6). The actual improvement is defined as the difference between the surrogate model evaluated at the candidate $s_\ell + p_\ell$ and the surrogate model evaluated at the current iterate s_ℓ. The expected improvement is defined as the value of the reduced surrogate model evaluated at r_ℓ. If the ratio is larger than a threshold, the candidate step p_ℓ is accepted and the trust-region radius is increased. Else, the candidate is rejected and the trust-region radius is decreased. The process is repeated until convergence criteria are met.

Algorithm 6: BNTR algorithm

Input: Initial guess s_0, $\Delta_0^{sub} > 0$

Take a finite number of gradient descent steps and update s_0, Δ_0^{sub}

for ℓ=*0,1,2,...* **do**

  Create active set of bounds $\mathscr{A}(s)$ and set of inactive directions $\mathscr{I}(s)$

  Construct reduced model gradient and reduced model Hessian based on $\mathscr{I}(s)$

  Solve for the optimum of the reduced model M_ℓ^r: $r \approx arg\ \min_r M_\ell^r(r)$ s.t. $r \leq \Delta_\ell^{sub}$

  Construct new candidate step p_ℓ by projecting r_ℓ onto $\mathscr{A}(s)$

  Calculate $\kappa_\ell = \frac{M_\ell^s(s_\ell + p_\ell) - M_\ell^s(s_\ell)}{M_\ell^r(r_\ell)}$, the ratio of actual over expected improvement

  if κ_ℓ *is larger than a threshold* **then**

    Accept step $s_{\ell+1} = s_\ell + p_\ell$

    Expand trust-region radius: $\Delta_{\ell+1}^{sub} = \alpha^{inc} \Delta_\ell^{sub}$

  else

    Reject step $s_{\ell+1} = s_\ell$

    Shrink trust-region radius $\Delta_{\ell+1}^{sub} = \alpha^{dec} \Delta_\ell^{sub}$

  Check convergence criteria

end

Appendix C Power Analysis

In this section, we derive the optimization problem we solve in the noisy acceptance step to determine the optimal number of objective function evaluations. For a more detailed discussion of power analysis, we refer to Montgomery (2008) or Cohen (1988).

C.1 Statistical Motivation

Suppose we observe two samples $\{y_1^{(1)},\dots,y_{n_1}^{(1)}\}$ and $\{y_1^{(2)},\dots,y_{n_2}^{(2)}\}$. The first sample has a mean of $\mathbb{E}[y_i^{(1)}] = \mu_1$ and the second a mean of $\mathbb{E}[y_i^{(2)}] = \mu_2$. We assume that both samples are independent of another and that the variance of both samples is σ^2. Assume further that the sample averages can be approximated by normal distributions, i.e.,

$$\frac{1}{n_1}\sum_{i=1}^{n_1} y_i^{(1)} \approx N(\mu_1, \sigma^2/n_1) \quad \text{and} \quad \frac{1}{n_2}\sum_{i=1}^{n_2} y_i^{(2)} \approx N(\mu_2, \sigma^2/n_2)$$

Note that this can be justified by distribution assumptions on the $y_i^{(k)}$ or by asymptotic arguments. In particular, we do not want to assume that the variables in a sample are independent, i.e., $y_i^{(k)}$ and $y_j^{(k)}$ may be dependent.

Our target parameter is $\Delta := \mu_1 - \mu_2$, and our goal is to test whether this parameter is greater than zero: $\Delta > 0$. The key assumption underlying power analysis is that we can choose the values of n_1 and n_2.

For this, we first need to select a statistical significance level $\alpha \in (0,1)$, a power level $1-\beta \in (0,1)$, and a minimal detectable effect size Δ_{min}. The formal test is then

$$H_0 : \Delta = 0 \text{ v.s. } H_1 : \Delta = \Delta_{min}$$

Define the estimators $\hat{\mu}_1 := \frac{1}{n_1}\sum_{i=1}^{n_1} y_i^{(1)}$ and $\hat{\mu}_2 := \frac{1}{n_2}\sum_{i=1}^{n_2} y_i^{(2)}$ of μ_1 and μ_2, respectively, and the estimator $\hat{\Delta} = \hat{\Delta}(n_1, n_2) := \hat{\mu}_1 - \hat{\mu}_2$. Under the normality assumption stated above, we get

$$\hat{\Delta}(n_1, n_2) \approx N(\Delta, \sigma_\Delta^2)$$

With $\sigma_\Delta^2 = \sigma^2 \frac{n_1+n_2}{n_1 n_2}$. Define the t-test statistic $\hat{t} = \hat{\Delta}/\sigma_\Delta$.

Under the null hypothesis H_0, we find $\hat{t} \approx N(0,1)$, so that we can choose the critical value, i.e., the value such that the Type-1 error is α, as $\Phi^{-1}(1-\alpha)$. Note furthter that under the alternative hypothesis H_1, we have $\hat{t} - \Delta_{min}/\sigma_\Delta \approx N(0,1)$.

We also require that the Type-2 error is at most β, i.e., we want that the probability of accepting H_0, when H_1 is true, is at most β. More formally,

$$\begin{aligned}\beta &\geq \mathbb{P}[\hat{t} \leq z_{1-\alpha}|H_2] \\ &= \mathbb{P}[\hat{t} - \Delta_{min}/\sigma_\Delta \leq \Phi^{-1}(1-\alpha) - \Delta_{min}/\sigma_\Delta|H_2] \\ &\approx \Phi(\Phi^{-1}(1-\alpha) - \Delta_{min}/\sigma_\Delta)\end{aligned}$$

And so,

$$\begin{aligned}\Phi^{-1}(\beta) &\gtrsim \Phi^{-1}(1-\alpha) - \Delta_{min}/\sigma_\Delta \\ &= \Phi^{-1}(1-\alpha) - \Delta_{min}/\sigma\sqrt{\frac{n_1 n_2}{n_1+n_2}}\end{aligned}$$

Rearranging the previous equation then gives

$$\frac{n_1 n_2}{n_1+n_2} \geq \left[\frac{\Phi^{-1}(1-\alpha) + \Phi^{-1}(1-\beta)}{\Delta_{min}/\sigma}\right]^2 \tag{C.1}$$

C.2 Optimal sample sizes

Given the condition in Equation C.1, we want to minimize the total number of samples $n_1 + n_2$. However, the exact problem we face in the noisy acceptance step (Subsection 5.3.5) in *tranquilo* may slightly differ from Equation C.1, as there may already exist previous samples n_1^{exist} and n_2^{exist}.

In this case, we solve the following problem

$$\underset{n_1,n_2\in\mathbb{N}}{\text{minimize}}\, n_1 + n_2 \text{ s.t } \frac{(n_1+n_1^{exist})(n_2+n_2^{exist})}{n_1+n_1^{exist}+n_2+n_2^{exist}} \geq \left[\frac{\Phi^{-1}(1-\alpha) + \Phi^{-1}(1-\beta)}{\Delta_{min}/\sigma}\right]^2$$

References

Arnoud, Antoine, Fatih Guvenen, and Tatjana Kleineberg. 2019. "Benchmarking Global Optimizers." NBER Working Papers 26340. National Bureau of Economic Research, Inc. URL: https://ideas.repec.org/p/nbr/nberwo/26340.html. Tiktak. [9]

Augustin, F., and Youssef Marzouk. 2017. "A trust-region method for derivative-free nonlinear constrained stochastic optimization." (03): [14]

Berndt, Ernst R., Bronwyn Hall, Robert Hall, and Jerry Hausman. 1974. "Estimation and Inference in Nonlinear Structural Models." In *Annals of Economic and Social Measurement, Volume 3, number 4*. National Bureau of Economic Research, Inc, 653–65. URL: https://EconPapers.repec.org/RePEc:nbr:nberch:10206. [18]

Bertsekas, Dimitri P. 1982. "Projected Newton Methods for Optimization Problems with Simple Constraints." *SIAM Journal on Control and Optimization* 20 (2): 221–46. DOI: 10.1137/0320018. [60]

Briani, Matteo, Alvise Sommariva, and Marco Vianello. 2012. "Computing Fekete and Lebesgue points: Simplex, square, disk." *Journal of Computational and Applied Mathematics* 236 (9): 2477–86. DOI: https://doi.org/10.1016/j.cam.2011.12.006. [23]

Cartis, Coralia, Jan Fiala, Benjamin Marteau, and Lindon Roberts. 2019. "Improving the Flexibility and Robustness of Model-Based Derivative-Free Optimization Solvers." *ACM Trans. Math. Softw.* 45 (3): DOI: 10.1145/3338517. [3, 5–7, 11, 13, 17, 18, 29, 31–33, 51]

Cartis, Coralia, and Lindon Roberts. 2019. "A derivative-free Gauss–Newton method." *Mathematical Programming Computation* 11 (4): 631–74. DOI: 10.1007/s12532-019-00161-7. [11, 31]

Cohen, J. 1988. *Statistical Power Analysis for the Behavioral Sciences*. Lawrence Erlbaum Associates. [62]

Conn, Andrew R., Nicholas I. M. Gould, and Philippe L. Toint. 2000. *Trust Region Methods*. Society for Industrial, and Applied Mathematics. DOI: 10.1137/1.9780898719857. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9780898719857. [5, 6, 10–12, 23, 59]

Dener, Alp, Adam Denchfield, Hansol Suh, Todd Munson, Jason Sarich, Stefan Wild, Steven Benson, and Lois Curfman McInnes. 2021. "TAO Users Manual (Rev. 3.15)." (3): DOI: 10.2172/1814593. [13, 30, 31, 60]

Dolan, Elizabeth D., and Jorge J. Moré. 2002. "Benchmarking optimization software with performance profiles." *Mathematical Programming* 91: 201–13. [31]

Eisenhauer, Philipp, James J. Heckman, and Stefano Mosso. 2015. "ESTIMATION OF DYNAMIC DISCRETE CHOICE MODELS BY MAXIMUM LIKELIHOOD AND THE SIMULATED METHOD OF MOMENTS." *International Economic Review* 56 (2): 331–57. DOI: https://doi.org/10.1111/iere.12107. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/iere.12107. [1, 7]

Gabler, Janoś. 2022. "A Python Tool for the Estimation of large scale scientific models." URL: https://github.com/OpenSourceEconomics/estimagic. [6]

Gabler, Janoś, Sebastian Gsell, Tim Mensinger, and Mariam Petrosyan. 2024. "Tranquilo." URL: https://github.com/OpenSourceEconomics/tranquilo. [6]

Gabler, Janoś, Tobias Raabe, Klara Röhrl, and Hans-Martin von Gaudecker. 2022. "The effectiveness of testing, vaccinations and contact restrictions for containing the CoViD-19 pandemic." en. *Sci. Rep.* 12 (1): 8048. [5]

Gould, Nicholas I. M., Dominique Orban, and Philippe L. Toint. 2003. "CUTEr and SifDec: A Constrained and Unconstrained Testing Environment, Revisited." *ACM Trans. Math. Softw.* 29 (4): 373–94. DOI: 10.1145/962437.962439. [31]

Larson, Jeffrey, Matt Menickelly, and Stefan M. Wild. 2019. "Derivative-free optimization methods." *Acta Numerica* 28 (5): 287–404. DOI: 10.1017/s0962492919000060. [10–12, 17, 26]

Lee, Donghoon, and Matthew Wiswall. 2007. "A Parallel Implementation of the Simplex Function Minimization Routine." *Computational Economics* 30 (02): 171–87. DOI: 10.1007/s10614-007-9094-2. [3, 14]

Levenberg, Kenneth. 1944. "A Method for the Solution of Certain Non-Linear Problems in Least Squares." *Quarterly of Applied Mathematics* 2 (2): 164–68. URL: http://www.jstor.org/stable/43633451 (visited on 01/09/2024). [3]

Marquardt, Donald W. 1963. "An Algorithm for Least-Squares Estimation of Nonlinear Parameters." *Journal of the Society for Industrial and Applied Mathematics* 11(2): 431–41. URL: http://www.jstor.org/stable/2098941 (visited on 01/09/2024). [3]

Montgomery, D.C. 2008. *Design and Analysis of Experiments*. Student solutions manual. John Wiley & Sons. URL: http://books.google.de/books?id=kMMJAm5bD34C. [62]

Moré, Jorge J., and Danny C. Sorensen. 1983. "Computing a Trust Region Step." *Siam Journal on Scientific and Statistical Computing* 4: 553–72. [58, 59]

Moré, Jorge J., and Stefan M. Wild. 2009. "Benchmarking Derivative-Free Optimization Algorithms." *SIAM Journal on Optimization* 20(1): 172–91. DOI: 10.1137/080724083. [6, 31, 32, 39]

Nocedal, Jorge, and Stephen Wright. 2006. *Numerical optimization*. Springer Science & Business Media. [10, 59]

Powell, M. 2009. "The BOBYQA Algorithm for Bound Constrained Optimization without Derivatives." Working paper DAMTP 2009/NA06. Centre for Mathematical Sciences, University of Cambridge. [6, 11–13, 17, 18, 26]

Powell, M. J. D. 2006. "The NEWUOA software for unconstrained optimization without derivatives." In. *Large-Scale Nonlinear Optimization*. Edited by G. Di Pillo and M. Roma. Boston, MA: Springer US, 255–97. DOI: 10.1007/0-387-30065-1_16. [11, 12, 26]

Pukelsheim, Friedrich. 2006. *Optimal Design of Experiments (Classics in Applied Mathematics) (Classics in Applied Mathematics, 50)*. USA: Society for Industrial, and Applied Mathematics. [22]

Shashaani, Sara, Fatemeh S. Hashemi, and Raghu Pasupathy. 2018. "ASTRO-DF: A Class of Adaptive Sampling Trust-Region Algorithms for Derivative-Free Stochastic Optimization." *SIAM Journal on Optimization* 28(4): 3145–76. DOI: 10.1137/15M1042425. eprint: https://doi.org/10.1137/15M1042425. [14]

Wild, Stefan M. 2008. "MNH: A Derivative-Free Optimization Algorithm Using Minimal Norm Hessians." In *Tenth Copper Mountain Conference on Iterative Methods*. URL: http://grandmaster.colorado.edu/~copper/2008/SCWinners/Wild.pdf. [26]

Wild, Stefan M. 2017. "Solving Derivative-Free Nonlinear Least Squares Problems with POUNDERS." In *Advances and Trends in Optimization with Engineering Applications*. Edited by Tamas Terlaky, Miguel F. Anjos, and Shabbir Ahmed. SIAM, 529–40. DOI: 10.1137/1.9781611974683.ch40. [3, 6, 13, 17–20, 26, 29–32]

Winfield, D. 1973. "Function Minimization by Interpolation in a Data Table." *IMA Journal of Applied Mathematics* 12(3): 339–47. DOI: 10.1093/imamat/12.3.339. eprint: https://academic.oup.com/imamat/article-pdf/12/3/339/1918553/12-3-339.pdf. [12]

Zhang, Hongchao, Andrew R. Conn, and Katya Scheinberg. 2010. "A Derivative-Free Algorithm for Least-Squares Minimization." *SIAM Journal on Optimization* 20(6): 3555–76. DOI: 10.1137/09075531X. [13, 18, 28]