Design Optimization Using Surrogates

Virginia Torczon
Department of Computer Science
College of William & Mary
and
ICASE
NASA Langley Research Center

Collaborators:

Andrew Booker, The Boeing Company
J. E. Dennis, Jr., Rice University
Paul Frank, The Boeing Company
David Serafini, E. O. Lawrence Berkeley National Lab
Michael Trosset, University of Arizona

The engineering design problem we will discuss:

Develop *computational* techniques for designing helicopter rotor blades that better minimize vibration transmitted to the hub.

The mathematical formulation:

A weighted sum of various functions of forces and moments.

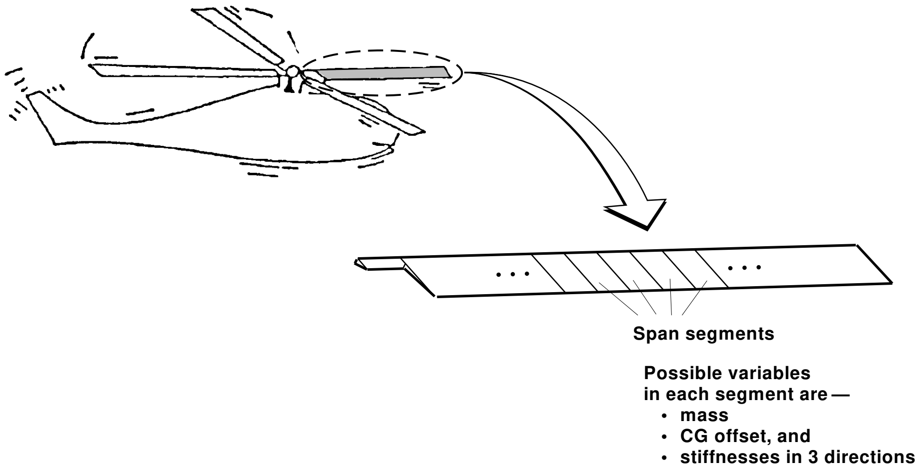
The analysis code:

Multidisciplinary, including dynamic structures, aerodynamics, wake modeling, and controls.

The design variables:

10 mass variables
10 centers of gravity
11 stiffnesses in a single direction

31 total design variables



The constraints:

Upper and lower bounds on all the design variables.
Upper bound on the sum of the masses.

So what makes this a challenging nonlinear optimization problem?

After all, it appears simple enough:

$$\begin{array}{rl} \text{minimize} & f(x) \\ \text{subject to} & a^T x \le c \\ & \ell \le x \le u, \end{array}$$

where x, a, ℓ, and $u \in \mathbf{R}^{31}$, $f : \mathbf{R}^{31} \rightarrow \mathbf{R}$ and $c \in \mathbf{R}$.

And why am I talking about this problem in a session with the title: Applications of Large-Scale Optimization?

There are only 31 design variables!

Features of the problem:

1. **Computational expense**.
   - execution time for a *fixed-wake analysis* varies from a few seconds to roughly twenty minutes on a mid-level workstation
   - execution time with *wake updating* may take several days on the same mid-level workstation

   Trade-off: increased computational cost associated with increased physical fidelity of the computation.

   Why is the computation so expensive?

   Large number of *state* (or *system*) variables.

   Bottom line: the objective is expensive to computed because the simulations upon which its value is based are expensive to compute. (Though the strategy we outline is amenable to other settings where the evaluation of the objective may be expensive for other reasons.)

2. **The objective function is nonsmooth with respect to the inputs**—due to the coupling of the disciplinary analysis codes.

3. **It is impractical to accurately approximate derivatives of the objective.**

4. **The values of the objective returned by the analysis code have few correct decimal digits.**

5. **If x violates any of the bound constraints, then $f(x)$ may not be available.**

6. **Even if x satisfies the explicit constraints, the analysis code may fail**—at the same cost as if a value had been obtained.

7. **The form of the objective function is subject to debate.**

   The engineers at Boeing Helicopter have what they consider to be a good design. They want to know if a better design *may* be possible, but they are not sure how to formulate what would constitute a better design.

   Consequence: the search must explore the design space; it is not sufficient to produce the "nearest" (local) minimizer.

Bottom line: This in not an optimization problem in the usual sense.

A single-minded pursuit of a minimizer is not desirable. Even if/when one is found:

- it may not be manufacturable,
- it may not prove to be better when *physical* tests are run, or
- the engineers may decide that they have the "wrong" objective.

Goals:

Given a modest number of evaluations of the objective (a "budget") produce:

1. a better understanding of the behavior of the objective function in the region(s) of interest and
2. do so in a way that is guaranteed—in theory—to converge to at least a local stationary point of the problem.

With a limited number of evaluations, it may not be possible to ensure convergence to a stationary point, but satisfying the conditions required by the theory provides some assurance that this is a sensible approach to the design optimization problem.

Fundamental idea:

Use a *sequence* of surrogate objective functions to *predict* optimal design candidates.

Exploit the flexibility of the analysis for *pattern search* methods to ensure global convergence.

Features of pattern search methods:

⋆ robust behavior in practice

⋆ observed behavior backed by convergence analysis

- do not require derivatives or approximations to derivatives
- simple to implement
- easy to use
- do not require numeric function values
- strong ties to experimental design and response surface methodology (RSM)
- straightforward to customize

Why so popular in the user community?

Applicable when:

- f is nondifferentiable

- derivatives are either not available or not reliable

- evaluation of f is inaccurate

- f is highly nonlinear

- only subjective (rank or order) information is available

- want to explore the design space (concerned about getting trapped at local minimizers)

Generalized pattern search:

Given $x_0 \in \mathbf{R}^n$, $f(x_0)$, $P_0 \in \mathbf{R}^{n \times p}$, and $\Delta_0 > 0$, for $k = 0, 1, \ldots$ until *convergence* do $\{$

1. Find a step s_k using `Exploratory Moves`(Δ_k,P_k).

2. If $f(x_k + s_k)$ is better than $f(x_k)$, then

$$x_{k+1} \quad = \quad x_k + s_k.$$

   Otherwise,

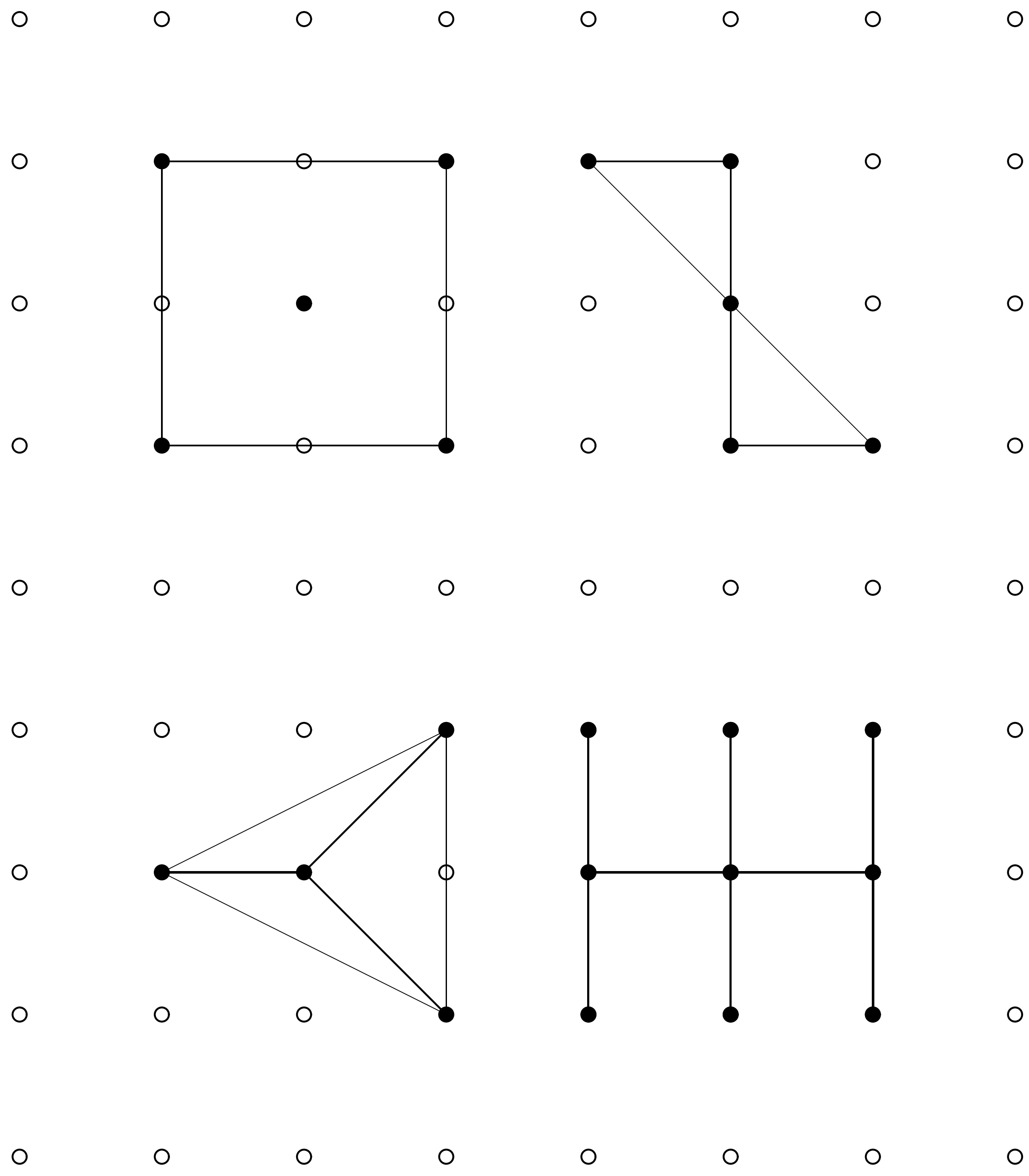
$$x_{k+1} \quad = \quad x_k.$$

3. `Update`(Δ_k,P_k)

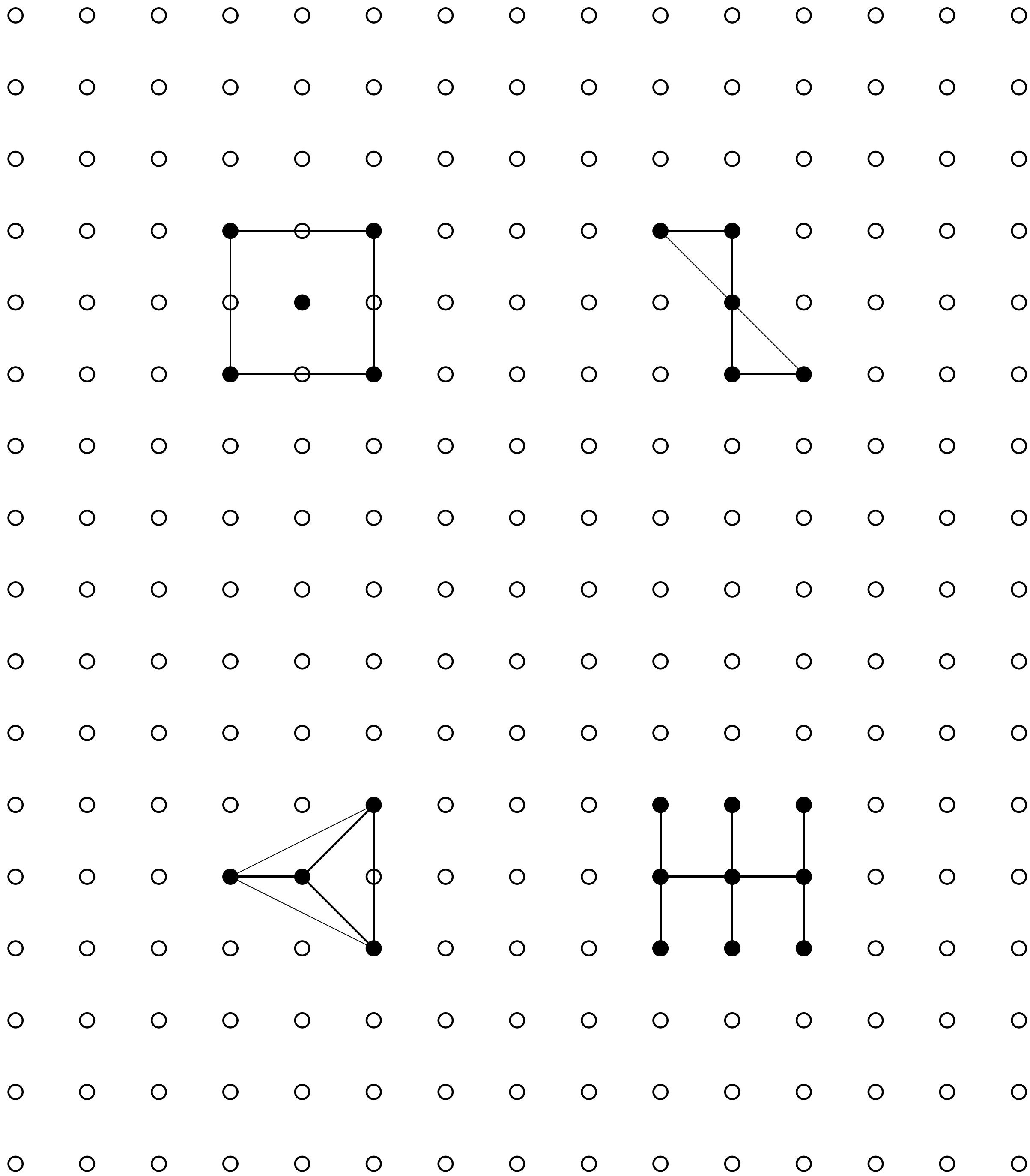
$\}$

First key feature of the analysis: the lattice

First key feature of the analysis: the lattice



The effect of reducing Δ:



Second key feature of the analysis:

Hypotheses on the outcome of the `Exploratory Moves`.

- Any trial step s_k^i *must* satisfy: $s_k^i \in \Delta_k P_k$ (i.e., the trial steps *must* be defined by the pattern).

- *Any* step $s_k^i \in \Delta_k P_k$ for which $f(x_k + s_k^i)$ is better than $f(x_k)$ is acceptable.

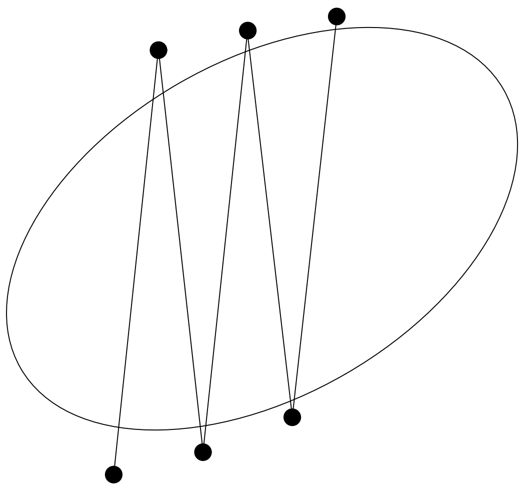
- Must have a "fall back" strategy based on using a positive basis for $\mathbf{R}^n$.

- Keep x_k—and reduce Δ_k—*only* if no step s_k^i defined by the fall back strategy satisfies the condition that $f(x_k + s_k^i)$ is better than $f(x_k)$.

Note:

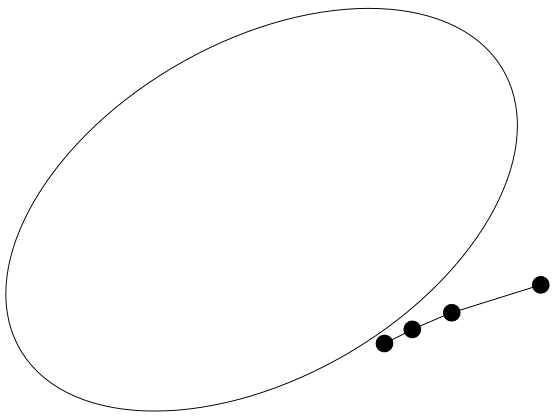
The pattern P_k must satisfy certain mild algebraic and geometric conditions, as must the update of Δ_k.

These two key features prevent the following:



decrease too small
relative to the
length of the step

$\Longrightarrow$ need for step length control



decrease too small
relative to the
norm of the gradient

Algorithmic features:

- admit *wide* range of heuristics—both old and new—and yet preserve robustness
  (1997 Torczon/Trosset, in revision for *Technometrics*)
  (1998 Booker/Dennis/Frank/Serafini/Torczon/Trosset, to appear in *Structural Optimization*)

- can be quite frugal:
  - best case, 1 evaluation of f/iteration;
  - worst case (the fall back strategy), anywhere from $(n+1)$—$2n$ evaluations of f/iteration

  (Lewis/Torczon, in revision for *Mathematical Programming*)

- embarrassingly parallel
  (1991 Dennis/Torczon, SIAM J. on Optimization)

Basic strategy:

We can choose *any* point on the current lattice that produces decrease on $f(x_k)$.

Use a *surrogate* of f to *predict* which points on the lattice are likely to predict decrease.

If the surrogate's prediction is good, we will only require *one* evaluation of f at the candidate solution.

The fall back strategy, which we either

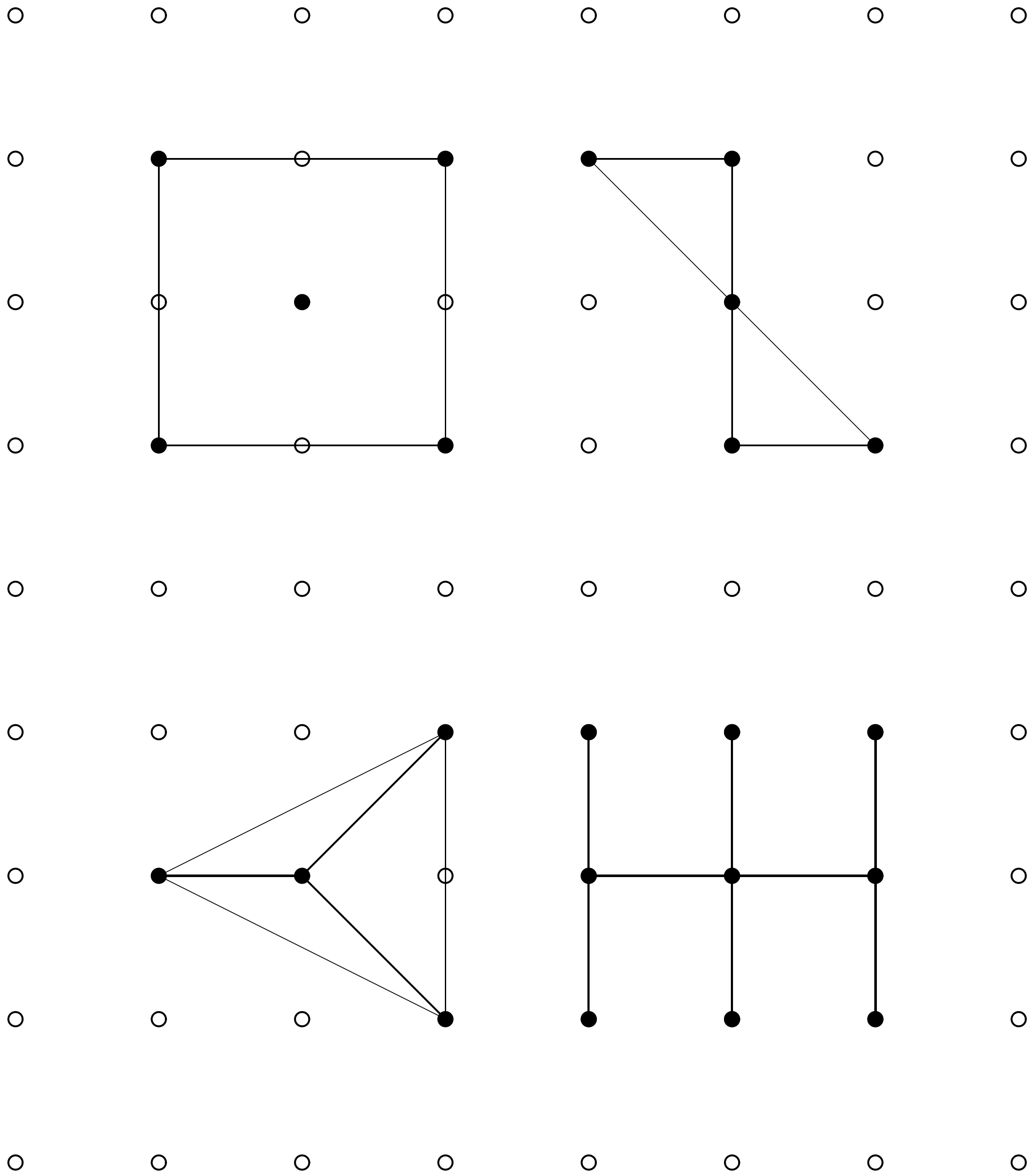
- *may* invoke—when the surrogate's predictions are deemed too poor or

- *must* invoke—when we need to reduce Δ_k

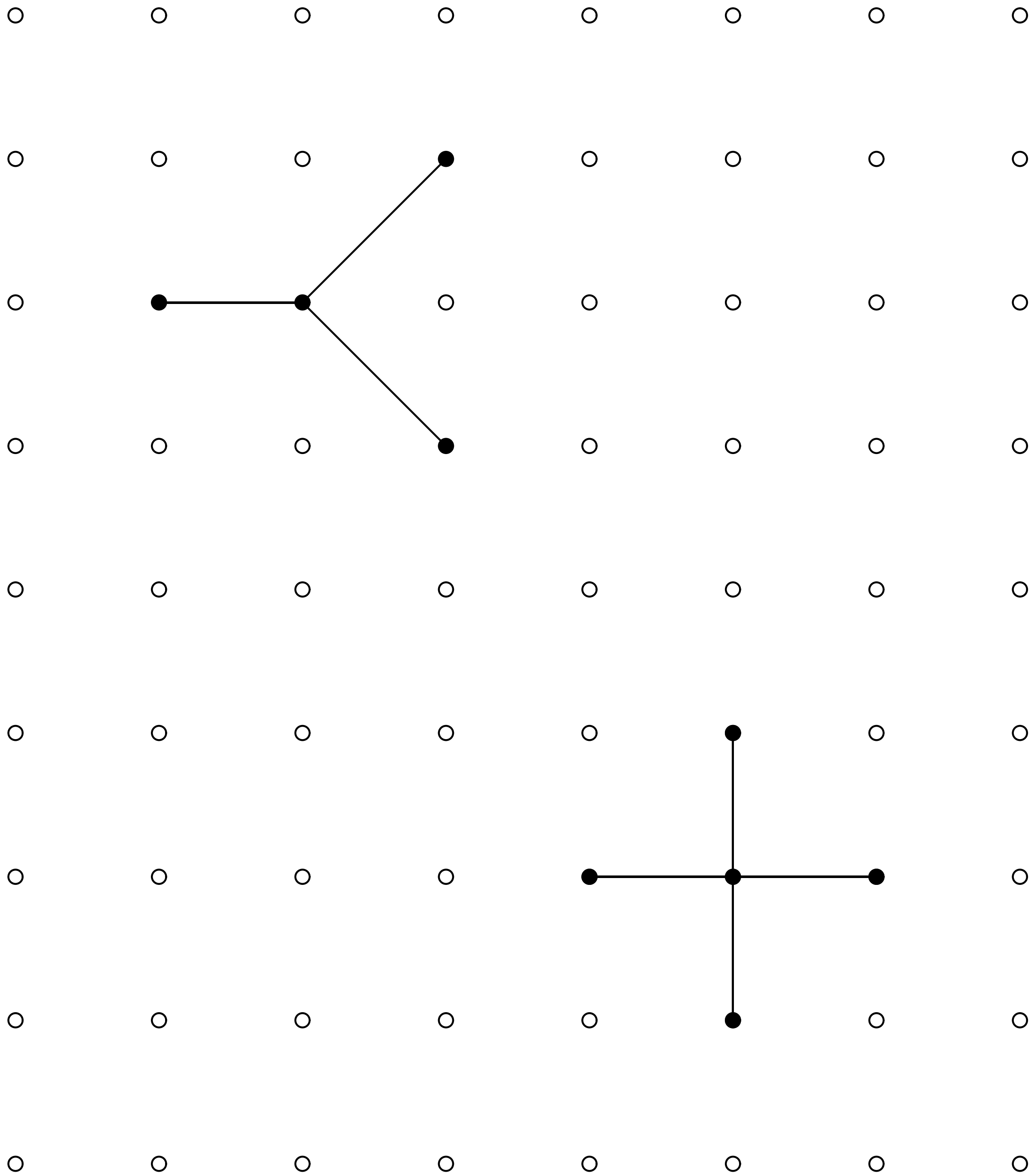
ensure global convergence.

The candidates:

Classical *static* sampling strategies:



Examples of fall back sampling strategies:



Approximations versus models:

Our basic strategy should work with any type of surrogate for the objective.

We distinguish between:

- *models*, which are auxiliary simulations that are less physically faithful, but also less computationally expensive, and

- *approximations*, which are algebraic summaries obtained from known values of the objective.

Our experience has been with approximations found in the design and analysis of computer experiments (DACE) literature, so we will focus on the use of approximations.

Basic strategy:

1. Choose:
   (a) an initial lattice over the feasible region
   (b) x_0

2. Perform an initial computer experiment:
   (a) select N initial design sites,
   (b) evaluate f at the initial design sites, and
   (c) construct an initial approximation a of f based on the values of f obtained at the design sites.

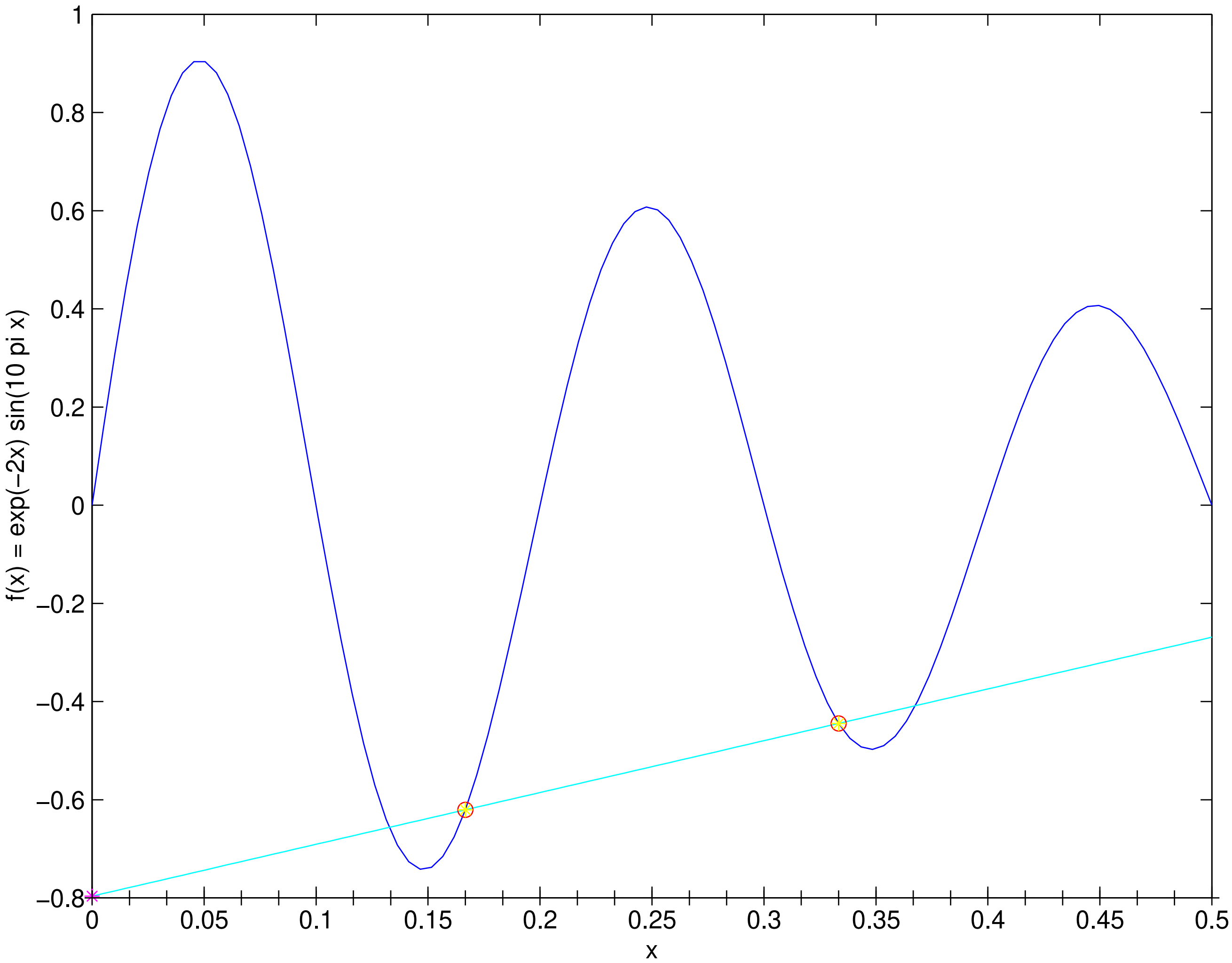
3. Do until the “budget” of evaluations has been exhausted (or an interesting design has been found):

   (a) find an $s_k \in \Delta_k P_k$ such that $a(x_k + s_k) < f(x_k)$ (i.e., a site at which a *predicts* improvement). Evaluate $f(x_k + s_k)$.
   (b) Update the approximation a to include the value $f(x_k + s_k)$.
   (c) If $f(x_k + s_k) < f(x_k)$ then let $x_{k+1} = x_k + s_k$. Else $x_{k+1} = x_k$
   (d) Repeat Step 3.

Simple example:

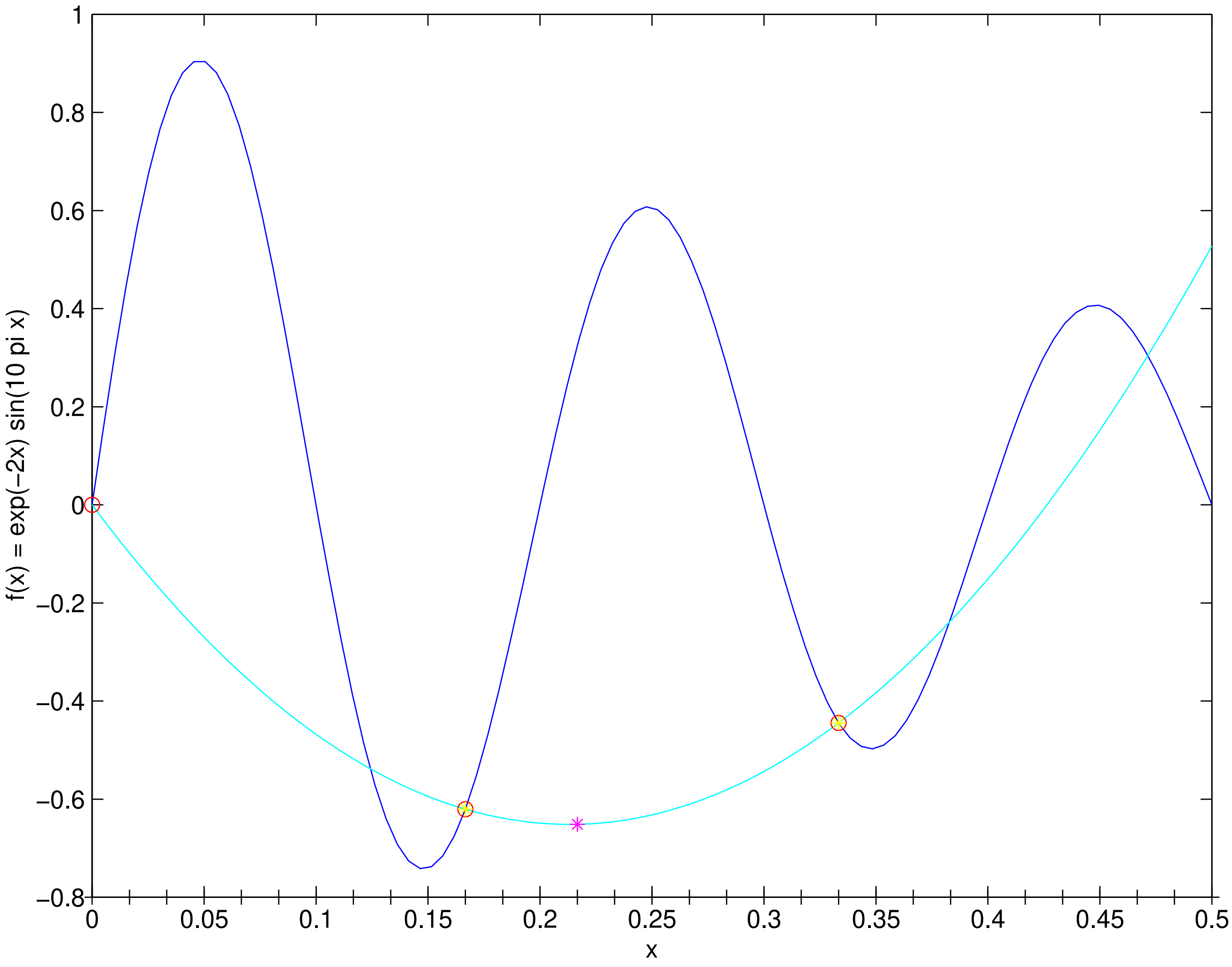
$$\begin{aligned} \text{minimize} \quad & f(x) = \exp^{-2x} \sin(10\pi x) \\ \text{subject to} \quad & 0 \leq x \leq 0.5, \end{aligned}$$

Iteration 1:



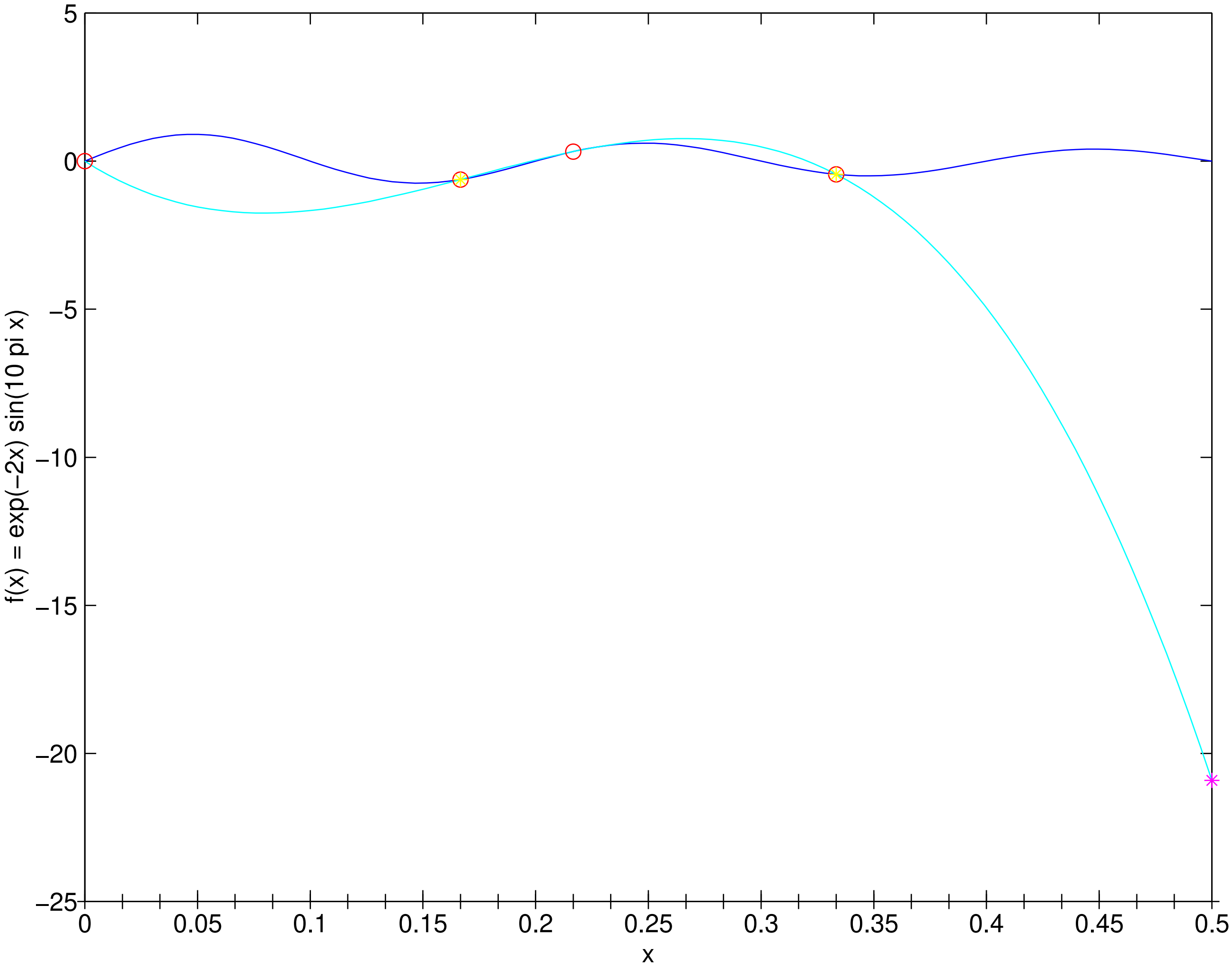
1
0.8
0.6
0.4
0.2
0
−0.2
−0.4
−0.6
−0.8
0
0.05
0.1
0.15
0.2
0.25
0.3
0.35
0.4
0.45
0.5
x
f(x) = exp(−2x) sin(10 pi x)

Iteration 2:



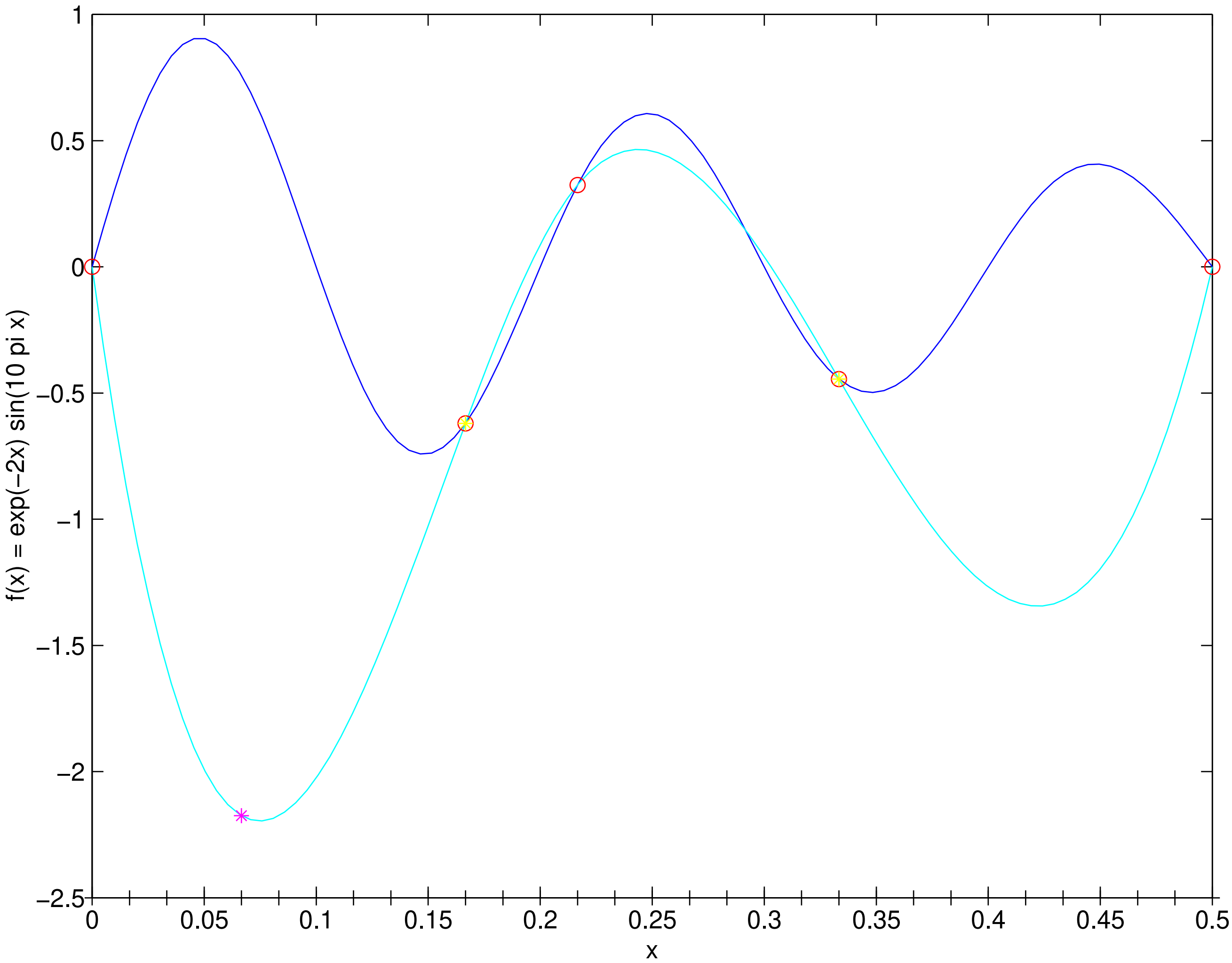
1
0.8
0.6
0.4
0.2
0
−0.2
−0.4
−0.6
−0.8
0
0.05
0.1
0.15
0.2
0.25
0.3
0.35
0.4
0.45
0.5
x
f(x) = exp(−2x) sin(10 pi x)

Iteration 3:

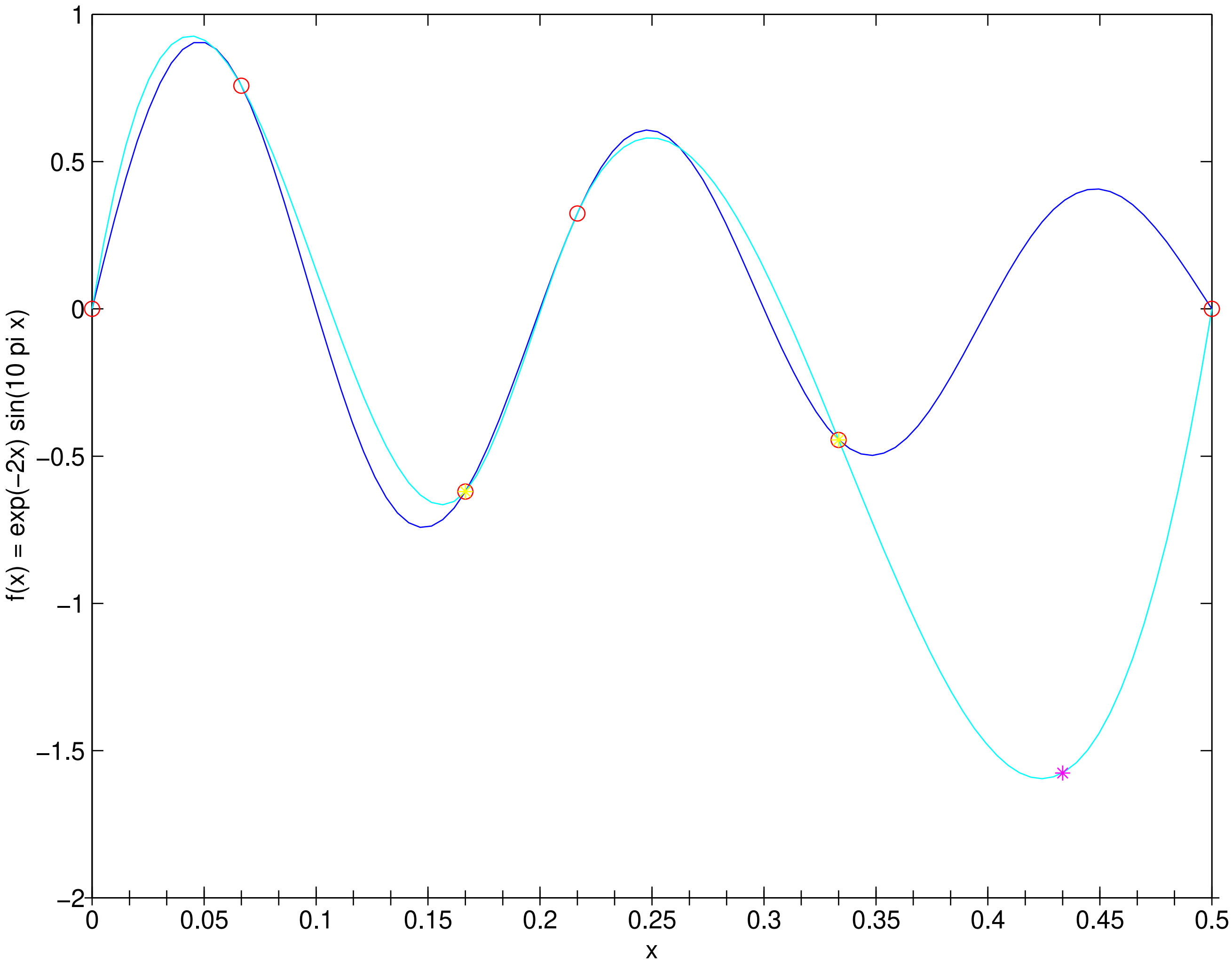


5
0
−5
−10
−15
−20
−25
0
0.05
0.1
0.15
0.2
0.25
0.3
0.35
0.4
0.45
0.5
x
f(x) = exp(−2x) sin(10 pi x)

Iteration 4:

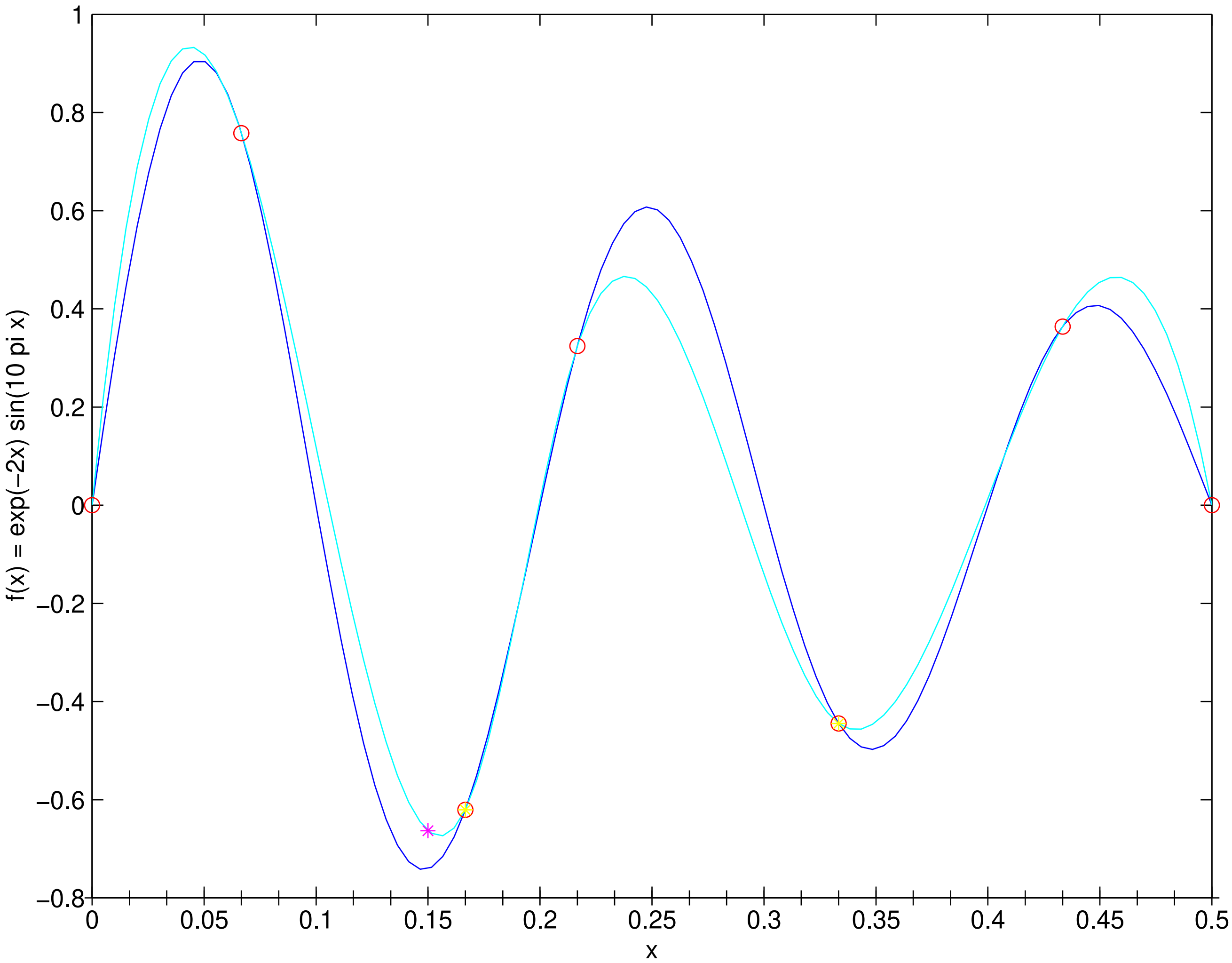

1
0.5
0
−0.5
−1
−1.5
−2
−2.5
f(x) = exp(−2x) sin(10 pi x)
0
0.05
0.1
0.15
0.2
0.25
0.3
0.35
0.4
0.45
0.5
x

Iteration 5:

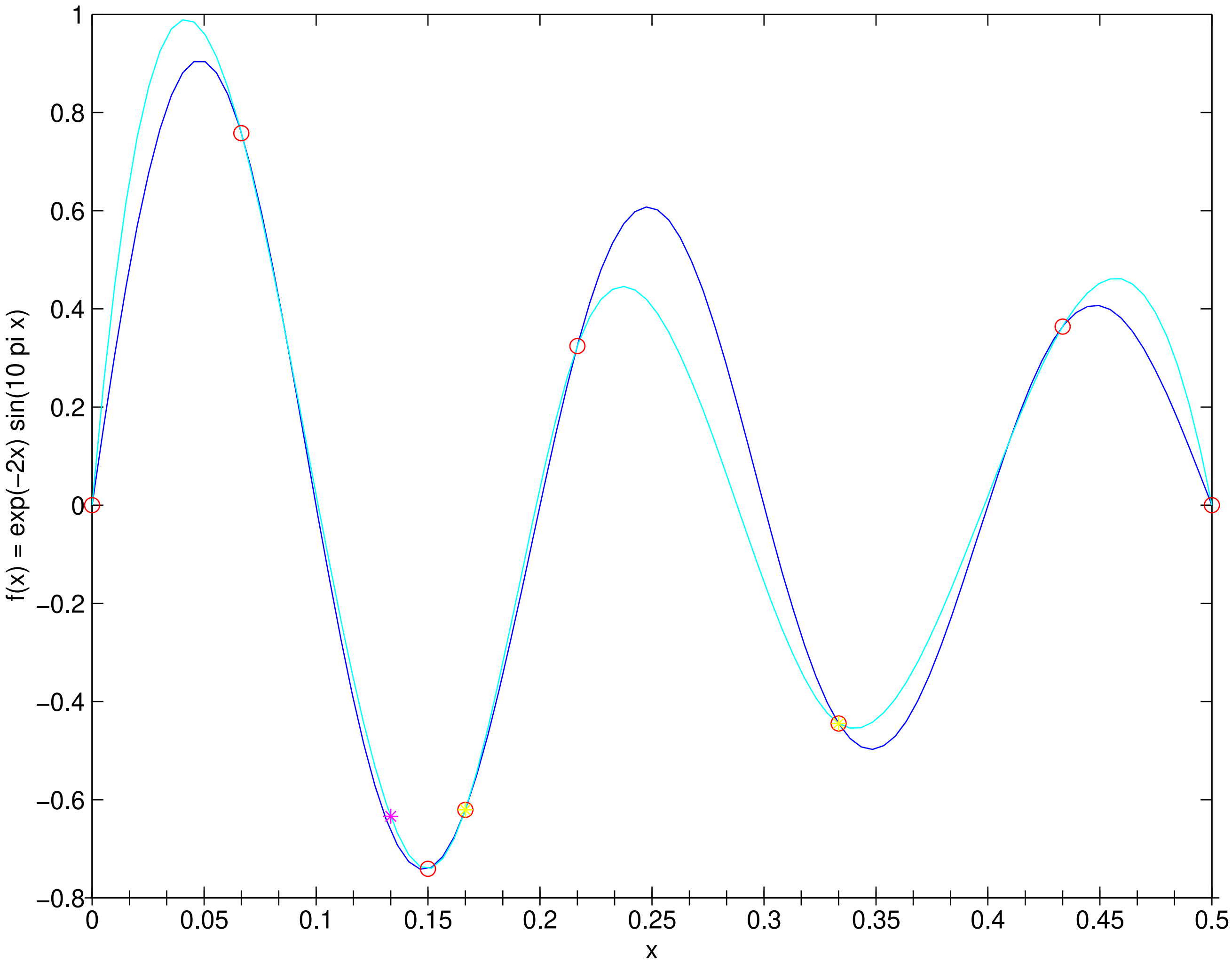


f(x) = exp(−2x) sin(10 pi x)
x
1
0.5
0
−0.5
−1
−1.5
−2
0
0.05
0.1
0.15
0.2
0.25
0.3
0.35
0.4
0.45
0.5

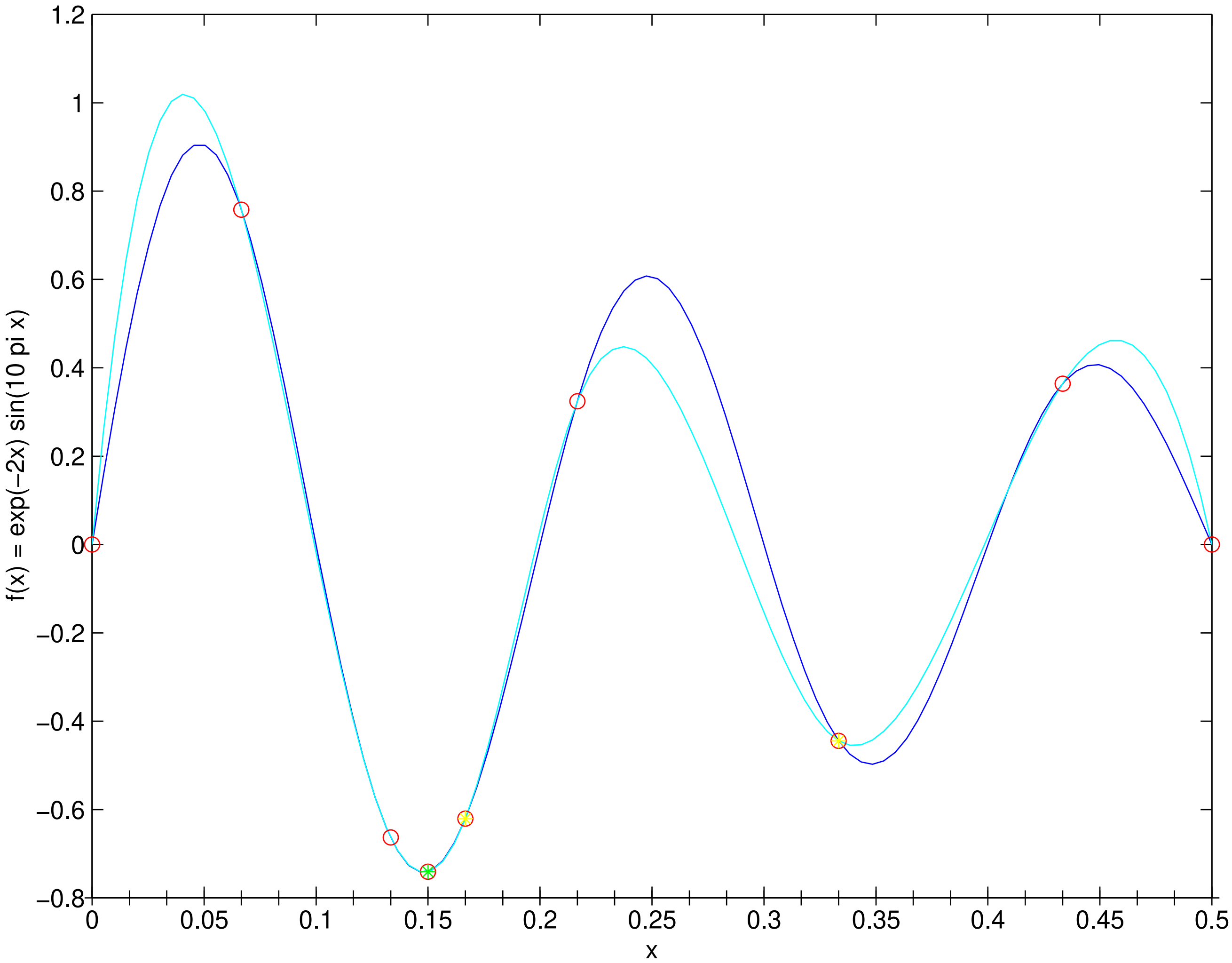
Iteration 6:


1
0.8
0.6
0.4
0.2
0
−0.2
−0.4
−0.6
−0.8
0
0.05
0.1
0.15
0.2
0.25
0.3
0.35
0.4
0.45
0.5
x
f(x) = exp(−2x) sin(10 pi x)

Iteration 7:


f(x) = exp(−2x) sin(10 pi x)
x
1
0.8
0.6
0.4
0.2
0
−0.2
−0.4
−0.6
−0.8
0
0.05
0.1
0.15
0.2
0.25
0.3
0.35
0.4
0.45
0.5

Iteration 8:



1.2
1
0.8
0.6
0.4
0.2
0
−0.2
−0.4
−0.6
−0.8
f(x) = exp(−2x) sin(10 pi x)
0
0.05
0.1
0.15
0.2
0.25
0.3
0.35
0.4
0.45
0.5
x

What have we gained?

Two levels of "optimization" at work:

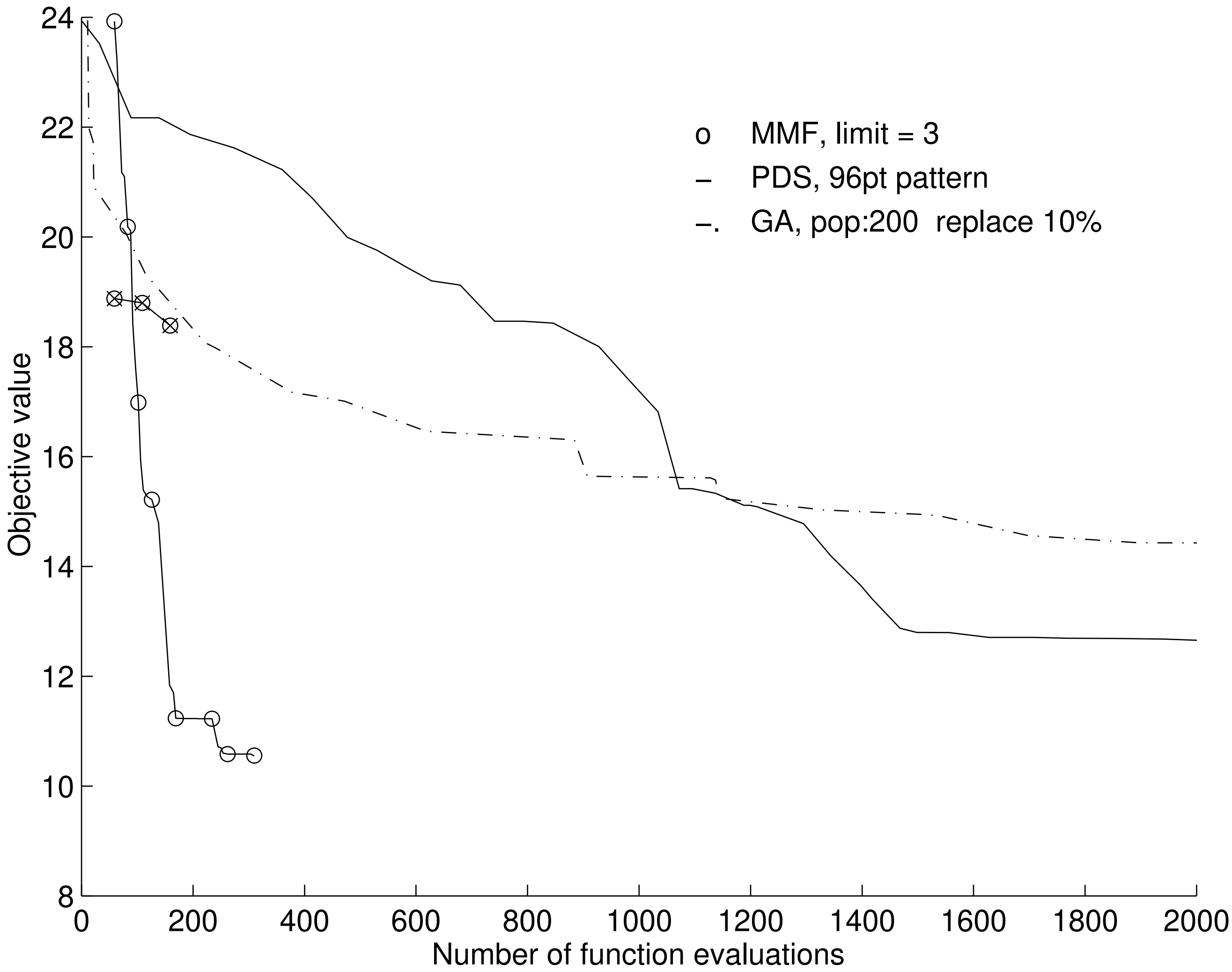
1. We "optimize" the current interpolant a_c. Since a_c is constructed to be inexpensive to compute, we can afford to expend as much effort as we deem necessary or appropriate.

2. We use the minimizer of the interpolant to predict a minimizer of the "true" objective f with the goal of expending only one new (expensive) function evaluation.

Notice that the initial interpolant does not necessarily have to be accurate for us to make progress. In the initial stages, the optimization exploits the weaknesses in the interpolant—by identifying those regions where we have the least information. But the sampling improves the interpolant, thus ultimately strengthening its predictive abilities.

But how does it work on more realistic problems?

Results for the 31-variable helicopter rotor blade design problem:



Future work:

- rigorous extensions to handle general constraints
- dealing with constraints that are expressed in terms of the outputs of computationally expensive analysis codes
- managing approximations more effectively as the optimization progresses
- balancing the trade-off between *optimization* and *exploration* more judiciously
- experimenting with more problems