simulannealbnd

Find minimum of function using simulated annealing algorithm

Syntax

```
x = simulannealbnd(fun,x0)
x = simulannealbnd(fun,x0,lb,ub)
x = simulannealbnd(fun,x0,lb,ub,options)
x = simulannealbnd(problem)
[x,fval] = simulannealbnd( ___ )
[x,fval,exitflag,output] = simulannealbnd( ___ )
```

Description

`x = simulannealbnd(fun,x0)` finds a local minimum, `x`, to the function handle `fun` that computes the values of the objective function. `x0` is an initial point for the simulated annealing algorithm, a real vector. example

> **Note**
>
> Passing Extra Parameters (Optimization Toolbox) explains how to pass extra parameters to the objective function, if necessary.

`x = simulannealbnd(fun,x0,lb,ub)` defines a set of lower and upper bounds on the design variables in `x`, so that the solution is always in the range `lb ≤ x ≤ ub`. If `x(i)` is unbounded below, set `lb(i) = -Inf`, and if `x(i)` is unbounded above, set `ub(i) = Inf`. example

`x = simulannealbnd(fun,x0,lb,ub,options)` minimizes with the optimization options specified in `options`. Create `options` using `optimoptions`. If no bounds exist, set `lb = []` and/or `ub = []`. example

`x = simulannealbnd(problem)` finds the minimum for `problem`, where `problem` is a structure described in . Create the `problem` structure by exporting a problem from Optimization app, as described in Exporting Your Work (Optimization Toolbox).

`[x,fval] = simulannealbnd( ___ )`, for any syntax, returns the value of the objective function `fun` at the solution `x`.

`[x,fval,exitflag,output] = simulannealbnd( ___ )` additionally returns a value `exitflag` that describes the exit condition of `simulannealbnd`, and a structure `output` with information about the optimization process. example

Examples

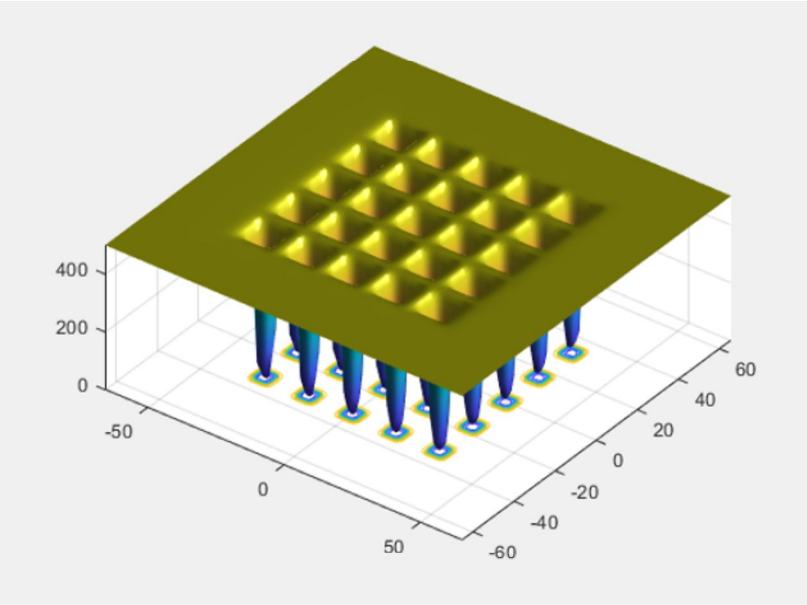
collapse all

Minimize a Function with Many Local Minima

Minimize De Jong's fifth function, a two-dimensional function with many local minima.

Plot De Jong's fifth function.

```
dejong5fcn
```



Minimize De Jong's fifth function using `simulannealbnd` starting from the point `[0,0]`.

```
fun = @dejong5fcn;
x0 = [0 0];
x = simulannealbnd(fun,x0)
```

```
Optimization terminated: change in best function value less than options.FunctionTolerance.
x = 1×2

   -32.0285   -0.1280
```

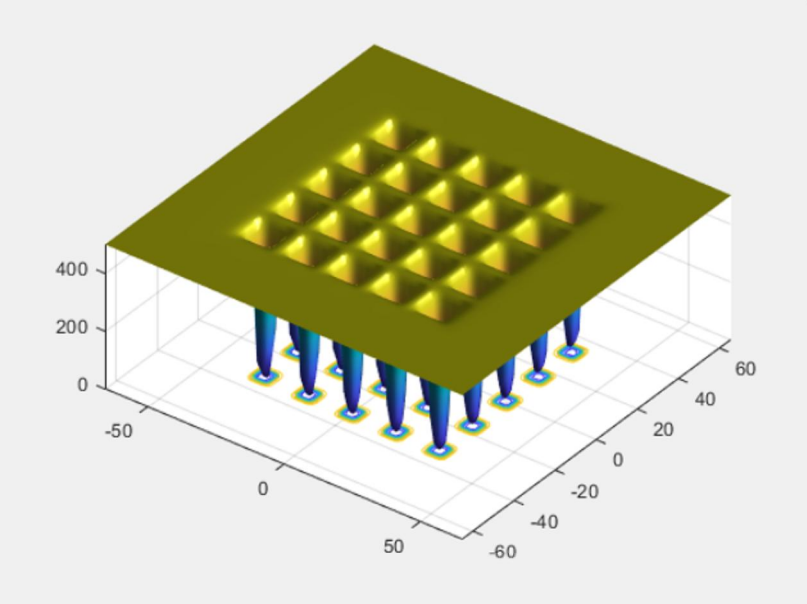
The `simulannealbnd` algorithm uses the MATLAB® random number stream, so you might obtain a different result.

Minimize Subject to Bounds

Minimize De Jong’ s fifth function within a bounded region.

Plot De Jong's fifth function.

```
dejong5fcn
```



Start `simulannealbnd` starting at the point `[0,0]`, and set lower bounds of -64 and upper bounds of 64 on each component.

```
fun = @dejong5fcn;
x0 = [0 0];
lb = [-64 -64];
ub = [64 64];
x = simulannealbnd(fun,x0,lb,ub)
```

```
Optimization terminated: change in best function value less than options.FunctionTolerance.
x = 1×2

   -15.9790  -31.9593
```

The `simulannealbnd` algorithm uses the MATLAB® random number stream, so you might obtain a different result.

Minimize Using Nondefault Options

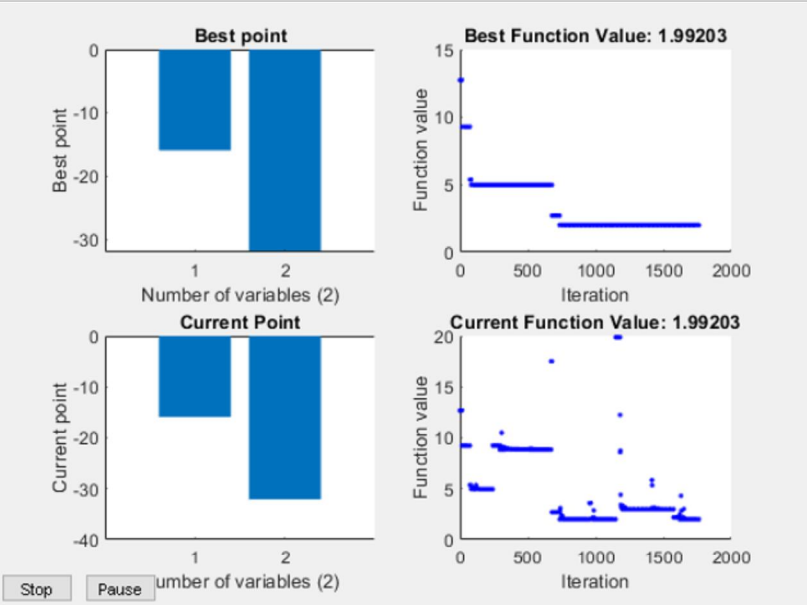
Observe the progress of `simulannealbnd` by setting options to use some plot functions.

Set simulated annealing options to use several plot functions.

```
options = optimoptions('simulannealbnd','PlotFcns',...
          {@saplotbestx,@saplotbestf,@saplotx,@saplotf});
```

Start `simulannealbnd` starting at the point `[0,0]`, and set lower bounds of -64 and upper bounds of 64 on each component.

```
rng default % For reproducibility
fun = @dejong5fcn;
x0 = [0,0];
lb = [-64,-64];
ub = [64,64];
x = simulannealbnd(fun,x0,lb,ub,options)
```



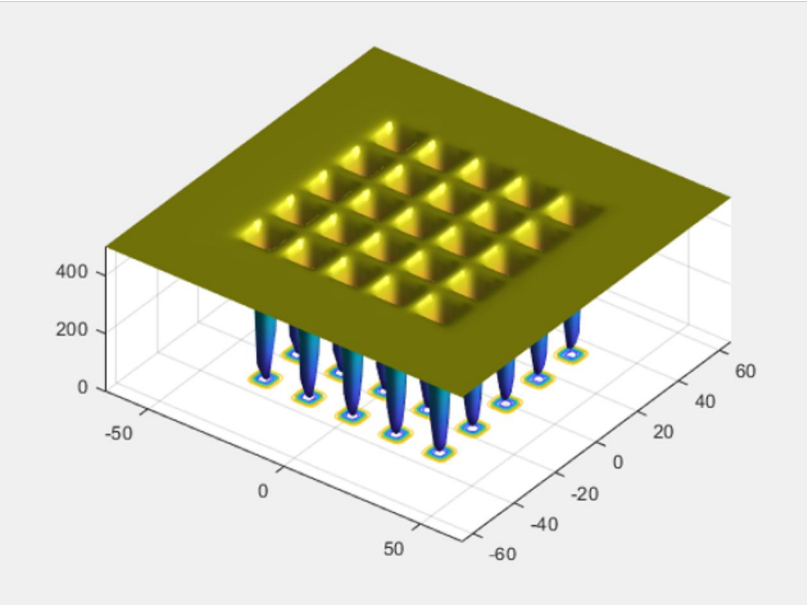

```
Optimization terminated: change in best function value less than options.FunctionTolerance.
x = 1×2

   -15.9790  -31.9593
```

Obtain All Outputs

Obtain all the outputs of a simulated annealing minimization.

Plot De Jong's fifth function.

```
dejong5fcn
```



Start `simulannealbnd` starting at the point `[0,0]`, and set lower bounds of -64 and upper bounds of 64 on each component.

```
fun = @dejong5fcn;
x0 = [0,0];
lb = [-64,-64];
ub = [64,64];
[x,fval,exitflag,output] = simulannealbnd(fun,x0,lb,ub)
```

```
Optimization terminated: change in best function value less than options.FunctionTolerance.
x = 1×2

  -15.9790  -31.9593

fval = 1.9920
exitflag = 1
output = struct with fields:
     iterations: 1762
      funccount: 1779
        message: 'Optimization terminated: change in best function value less than options.FunctionTolerance.'
       rngstate: [1x1 struct]
    problemtype: 'boundconstraints'
    temperature: [2x1 double]
      totaltime: 1.0315
```

The `simulannealbnd` algorithm uses the MATLAB® random number stream, so you might obtain a different result.

Input Arguments

collapse all

`fun` — Function to be minimized

function handle | function name

Function to be minimized, specified as a function handle or function name.`fun` is a function that accepts a vector `x` and returns a real scalar `f`, the objective function evaluated at `x`.

`fun` can be specified as a function handle for a file:

```
x = simulannealbnd(@myfun,x0)
```

where `myfun` is a MATLAB® function such as

```
function f = myfun(x)
f = ...            % Compute function value at x
```

`fun` can also be a function handle for an anonymous function:

```
x = simulannealbnd(@(x)norm(x)^2,x0,lb,ub);
```

Example: `fun = @(x)sin(x(1))*cos(x(2))`

Data Types: `char` | `function_handle` | `string`

`x0` — Initial point

real vector

Initial point, specified as a real vector. `simulannealbnd` uses the number of elements in `x0` to determine the number of variables that `fun` accepts.

Example: `x0 = [1,2,3,4]`

Data Types: `double`

`lb` — Lower bounds

real vector | real array

Lower bounds, specified as a real vector or real array. If the number of elements in `x0` is equal to that of `lb`, then `lb` specifies that

`x(i) >= lb(i)` for all `i`.

If `numel(lb) < numel(x0)`, then `lb` specifies that

`x(i) >= lb(i)` for `1 <= i <= numel(lb)`.

In this case, solvers issue a warning.

Example: To specify that all control variables are positive, `lb = zeros(size(x0))`

Data Types: `double`

`ub` — Upper bounds

real vector | real array

Upper bounds, specified as a real vector or real array. If the number of elements in `x0` is equal to that of `ub`, then `ub` specifies that

`x(i) <= ub(i)` for all `i`.

If `numel(ub) < numel(x0)`, then `ub` specifies that

`x(i) <= ub(i)` for `1 <= i <= numel(ub)`.

In this case, solvers issue a warning.

Example: To specify that all control variables are less than one, `ub = ones(size(x0))`

Data Types: `double`

`options` — Optimization options

object returned by `optimoptions` | structure

Optimization options, specified as an object returned by `optimoptions` or a structure. For details, see Simulated Annealing Options.

`optimoptions` hides the options listed in *italics*; see Options that optimoptions Hides.

`{}` denotes the default value. See option details in Simulated Annealing Options.

Option	Description	Values
`AcceptanceFcn`	Function the algorithm uses to determine if a new point is accepted. Specify as `'acceptancesa'` or a function handle.	Function handle | `{'acceptancesa'}`

Option	Description	Values
`AnnealingFcn`	Function the algorithm uses to generate new points. Specify as a name of a built-in annealing function or a function handle.	Function handle | function name | `'annealingboltz'` | `{'annealingfast'}`
`DataType`	Type of decision variable	`'custom'` | `{'double'}`
`Display`	Level of display	`'off'` | `'iter'` | `'diagnose'` | `{'final'}`
`DisplayInterval`	Interval for iterative display	Positive integer | `{10}`
`FunctionTolerance`	Termination tolerance on function value For an options structure, use `TolFun`.	Positive scalar | `{1e-6}`
`HybridFcn`	Automatically run `HybridFcn` (another optimization function) during or at the end of iterations of the solver. Specify as a name or a function handle. See When to Use a Hybrid Function.	`'fminsearch'` | `'patternsearch'` | `'fminunc'` | `'fmincon'` | `{[]}` or 1-by-2 cell array | `{@solver, hybridoptions}`, where `solver = fminsearch`, `patternsearch`, `fminunc`, or `fmincon` `{[]}`
`HybridInterval`	Interval (if not `'end'` or `'never'`) at which `HybridFcn` is called	Positive integer | `'never'` | `{'end'}`
`InitialTemperature`	Initial value of temperature	Positive scalar | positive vector | `{100}`
`MaxFunctionEvaluations`	Maximum number of objective function evaluations allowed For an options structure, use `MaxFunEvals`.	Positive integer | `{3000*numberOfVariables}`
`MaxIterations`	Maximum number of iterations allowed For an options structure, use `MaxIter`.	Positive integer | `{Inf}`
`MaxStallIterations`	Number of iterations over which average change in fitness function value at current point is less than `options.FunctionTolerance` For an options structure, use `StallIterLimit`.	Positive integer | `{500*numberOfVariables}`
`MaxTime`	The algorithm stops after running for `MaxTime` seconds For an options structure, use `TimeLimit`.	Positive scalar | `{Inf}`
`ObjectiveLimit`	Minimum objective function value desired	Scalar | `{-Inf}`
`OutputFcn`	Function(s) get(s) iterative data and can change options at run time For an options structure, use `OutputFcns`.	Function handle | cell array of function handles | `{[]}`
`PlotFcn`	Plot function(s) called during iterations For an options structure, use `PlotFcns`.	Function handle | built-in plot function name | cell array of function handles | cell array of built-in plot function names | `'saplotbestf'` | `'saplotbestx'` | `'saplotf'` | `'saplotstopping'` | `'saplottemperature'` | `{[]}`
`PlotInterval`	Plot functions are called at every interval	Positive integer | `{1}`
`ReannealInterval`	Reannealing interval	Positive integer | `{100}`
`TemperatureFcn`	Function used to update temperature schedule	Function handle | built-in temperature function name | `'temperatureboltz'` | `'temperaturefast'` | `{'temperatureexp'}`

Example: `options = optimoptions(@simulannealbnd,'MaxIterations',150)`

Data Types: `struct`

`problem` — Problem structure

structure

Problem structure, specified as a structure with the following fields:

- `objective` — Objective function
- `x0` — Starting point
- `lb` — Lower bound for `x`
- `ub` — Upper bound for `x`
- `solver` — `'simulannealbnd'`
- `options` — Options created with `optimoptions` or an options structure
- `rngstate` — Optional field to reset the state of the random number generator

Create the structure `problem` by exporting a problem from the Optimization app, as described in Importing and Exporting Your Work (Optimization Toolbox).

> **Note**
>
> `problem` must have all the fields as specified above.

Data Types: `struct`

Output Arguments

collapse all

`x` — Solution

real vector

Solution, returned as a real vector. The size of `x` is the same as the size of `x0`. Typically, `x` is a local solution to the problem when `exitflag` is positive.

`fval` — Objective function value at the solution

real number

Objective function value at the solution, returned as a real number. Generally, `fval = fun(x)`.

`exitflag` — Reason `simulannealbnd` stopped

integer

Reason `simulannealbnd` stopped, returned as an integer.

Exit Flag	Meaning
`1`	Average change in the value of the objective function over `options.MaxStallIterations` iterations is less than `options.FunctionTolerance`.
`5`	Objective function value is less than `options.ObjectiveLimit`.
`0`	Maximum number of function evaluations or iterations reached.
`-1`	Optimization terminated by an output function or plot function.
`-2`	No feasible point found.
`-5`	Time limit exceeded.

`output` — Information about the optimization process

structure

Information about the optimization process, returned as a structure with fields:

- `problemtype` — Type of problem: unconstrained or bound constrained.

- `iterations` — The number of iterations computed.
- `funccount` — The number of evaluations of the objective function.
- `message` — The reason the algorithm terminated.
- `temperature` — Temperature when the solver terminated.
- `totaltime` — Total time for the solver to run.
- `rngstate` — State of the MATLAB random number generator, just before the algorithm started. You can use the values in `rngstate` to reproduce the output of `simulannealbnd`. See Reproduce Your Results.

See Also

`ga` | `optimoptions` | `patternsearch`

Topics

Minimization Using Simulated Annealing Algorithm
Simulated Annealing Options
Multiprocessor Scheduling using Simulated Annealing with a Custom Data Type
Optimization Workflow
What Is Simulated Annealing?
Simulated Annealing Terminology
How Simulated Annealing Works

Introduced in R2007a

Simulated Annealing Options

This example shows how to create and manage options for the simulated annealing function `simulannealbnd` using `optimoptions` in the Global Optimization Toolbox.

View MATLAB Command

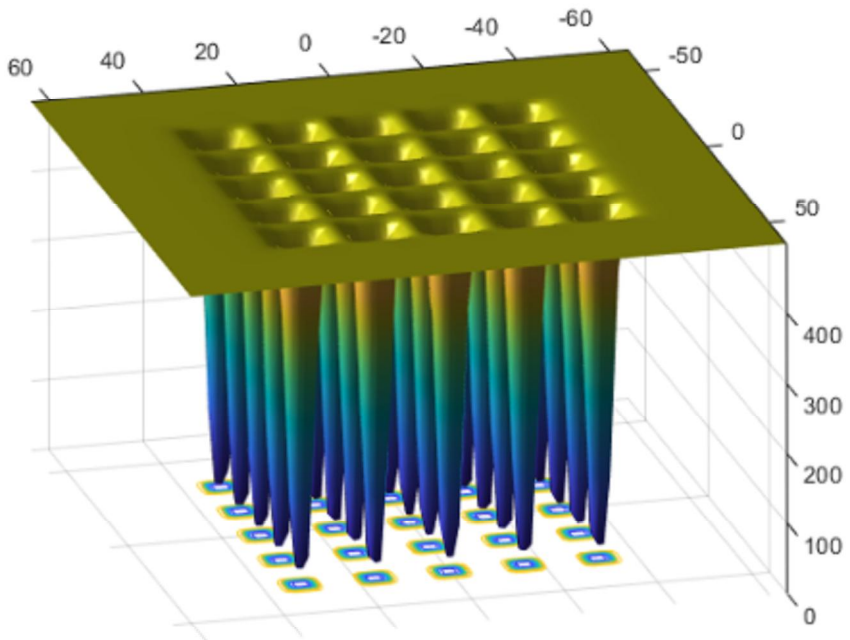
Optimization Problem Setup

`simulannealbnd` searches for a minimum of a function using simulated annealing. For this example we use `simulannealbnd` to minimize the objective function `dejong5fcn`. This function is a real valued function of two variables and has many local minima making it difficult to optimize. There is only one global minimum at `x =(-32,-32)`, where `f(x) = 0.998`. To define our problem, we must define the objective function, start point, and bounds specified by the range `-64 <= x(i) <= 64` for each `x(i)`.

```
ObjectiveFunction = @dejong5fcn;
startingPoint = [-30 0];
lb = [-64 -64];
ub = [64 64];
```

The function `plotobjective` in the toolbox plots the objective function over the range `-64 <= x1 <= 64`, `-64 <= x2 <= 64`.

```
plotobjective(ObjectiveFunction,[-64 64; -64 64]);
view(-15,150);
```



Now, we can run the `simulannealbnd` solver to minimize our objective function.

```
rng default % For reproducibility
[x,fval,exitFlag,output] = simulannealbnd(ObjectiveFunction,startingPoint,lb,ub);
```

```
Optimization terminated: change in best function value less than options.FunctionTolerance.
```

```
fprintf('The number of iterations was : %d\n', output.iterations);
```

```
The number of iterations was : 1095
```

```
fprintf('The number of function evaluations was : %d\n', output.funccount);
```

```
The number of function evaluations was : 1104
```

```
fprintf('The best function value found was : %g\n', fval);
```

```
The best function value found was : 2.98211
```

Note that when you run this example, your results may be different from the results shown above because simulated annealing algorithm uses random numbers to generate points.

Adding Visualization

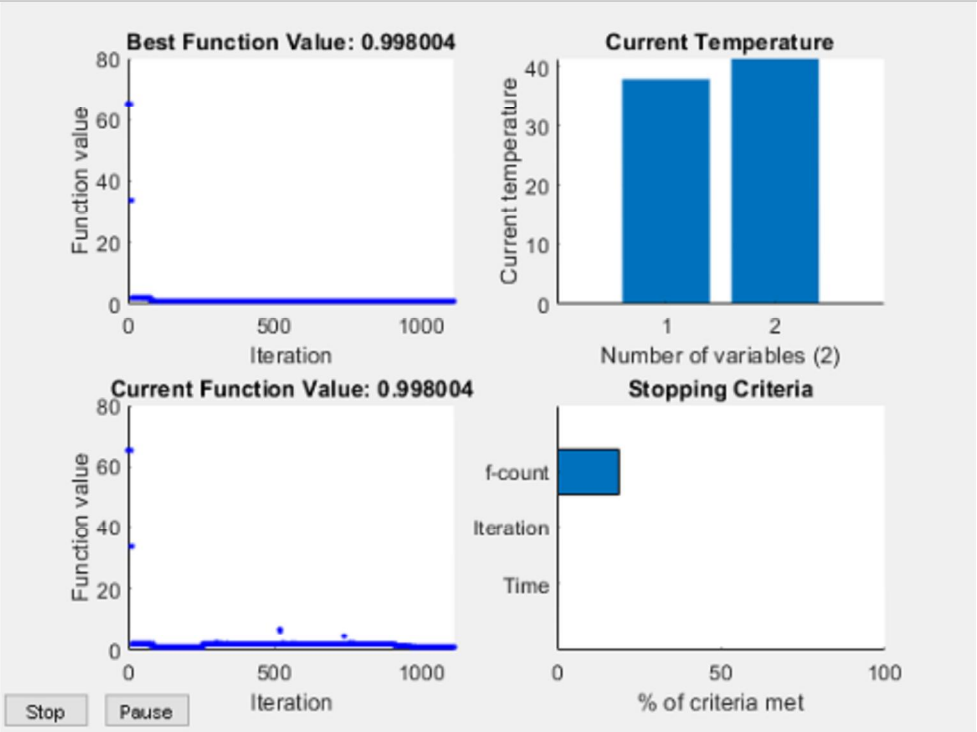
`simulannealbnd` can accept one or more plot functions through an 'options' argument. This feature is useful for visualizing the performance of the solver at run time. Plot functions are selected using `optimoptions`. The toolbox contains a set of plot functions to choose from, or you can provide your own custom plot functions.

To select multiple plot functions, set the `PlotFcn` option via the `optimoptions` function. For this example, we select `saplotbestf`, which plots the best function value every iteration, `saplottemperature`, which shows the current temperature in each dimension at every iteration, `saplotf`, which shows the current function value (remember that the current value is not necessarily the best one), and `saplotstopping`, which plots the percentage of stopping criteria satisfied every ten iterations.

```
options = optimoptions(@simulannealbnd, ...
                    'PlotFcn',{@saplotbestf,@saplottemperature,@saplotf,@saplotstopping});
```

Run the solver.

```
simulannealbnd(ObjectiveFunction,startingPoint,lb,ub,options);
```




```
Optimization terminated: change in best function value less than options.FunctionTolerance.
```

Specifying Temperature Options

The temperature parameter used in simulated annealing controls the overall search results. The temperature for each dimension is used to limit the extent of search in that dimension. The toolbox lets you specify initial temperature as well as ways to update temperature during the solution process. The two temperature-related options are the `InitialTemperature` and the `TemperatureFcn`.

Specifying initial temperature

The default initial temperature is set to 100 for each dimension. If you want the initial temperature to be different in different dimensions then you must specify a vector of temperatures. This may be necessary in cases when problem is scaled differently in each dimensions. For example,

```
options = optimoptions(@simulannealbnd,'InitialTemperature',[300 50]);
```

`InitialTemperature` can be set to a vector of length less than the number of variables (dimension); the solver expands the vector to the remaining dimensions by taking the last element of the initial temperature vector. Here we want the initial temperature to be the same in all dimensions so we need only specify the single temperature.

```
options.InitialTemperature = 100;
```

Specifying a temperature function

The default temperature function used by `simulannealbnd` is called `temperatureexp`. In the temperatureexp schedule, the temperature at any given step is .95 times the temperature at the previous step. This causes the temperature to go down slowly at first but ultimately get cooler faster than other schemes. If another scheme is desired, e.g. Boltzmann schedule or "Fast" schedule annealing, then `temperatureboltz` or `temperaturefast` can be used respectively. To select the fast temperature schedule, we can update our previously created options, changing `TemperatureFcn` directly.

```
options.TemperatureFcn = @temperaturefast;
```

Specifying reannealing

Reannealing is a part of annealing process. After a certain number of new points are accepted, the temperature is raised to a higher value in hope to restart the search and move out of a local minima. Performing reannealing too soon may not help the solver identify a minimum, so a relatively high interval is a good choice. The interval at which reannealing happens can be set using the `ReannealInterval` option. Here, we reduce the default reannealing interval to 50 because the function seems to be flat in many regions and solver might get stuck rapidly.

```
options.ReannealInterval = 50;
```

Now that we have setup the new temperature options we run the solver again

```
[x,fval,exitFlag,output] = simulannealbnd(ObjectiveFunction,startingPoint,lb,ub,options);
```

```
Optimization terminated: change in best function value less than options.FunctionTolerance.
```

```
fprintf('The number of iterations was : %d\n', output.iterations);
```

```
The number of iterations was : 1306
```

```
fprintf('The number of function evaluations was : %d\n', output.funccount);
```

```
The number of function evaluations was : 1321
```

```
fprintf('The best function value found was : %g\n', fval);
```

```
The best function value found was : 16.4409
```

Reproducing Results

`simulannealbnd` is a nondeterministic algorithm. This means that running the solver more than once without changing any settings may give different results. This is because `simulannealbnd` utilizes MATLAB® random number generators when it generates subsequent points and also when it determines whether or not to accept new points. Every time a random number is generated the state of the random number generators change.

To see this, two runs of `simulannealbnd` solver yields:

```
[x,fval] = simulannealbnd(ObjectiveFunction,startingPoint,lb,ub,options);
```

```
Optimization terminated: change in best function value less than options.FunctionTolerance.
```

```
fprintf('The best function value found was : %g\n', fval);
```

```
The best function value found was : 1.99203
```

And,

```
[x,fval] = simulannealbnd(ObjectiveFunction,startingPoint,lb,ub,options);
```

```
Optimization terminated: change in best function value less than options.FunctionTolerance.
```

```
fprintf('The best function value found was : %g\n', fval);
```

```
The best function value found was : 10.7632
```

In the previous two runs `simulannealbnd` gives different results.

We can reproduce our results if we reset the states of the random number generators between runs of the solver by using information returned by `simulannealbnd`. `simulannealbnd` returns the states of the random number generators at the time `simulannealbnd` is called in the output argument. This information can be used to reset the states. Here we reset the states between runs using this output information so the results of the next two runs are the same.

```
[x,fval,exitFlag,output] = simulannealbnd(ObjectiveFunction,startingPoint,lb,ub,options);
```

```
Optimization terminated: change in best function value less than options.FunctionTolerance.
```

```
fprintf('The best function value found was : %g\n', fval);
```

```
The best function value found was : 20.1535
```

We reset the state of the random number generator.

```
strm = RandStream.getGlobalStream;
strm.State = output.rngstate.State;
```

Now, let's run `simulannealbnd` again.

```
[x,fval] = simulannealbnd(ObjectiveFunction,startingPoint,lb,ub,options);
```

```
Optimization terminated: change in best function value less than options.FunctionTolerance.
```

```
fprintf('The best function value found was : %g\n', fval);
```

```
The best function value found was : 20.1535
```

Modifying the Stopping Criteria

`simulannealbnd` uses six different criteria to determine when to stop the solver. `simulannealbnd` stops when the maximum number of iterations or function evaluation is exceeded; by default the maximum number of iterations is set to Inf and the maximum number of function evaluations is `3000*numberOfVariables`. `simulannealbnd` keeps track of the average change in the function value for `MaxStallIterations` iterations. If the average change is smaller than the function tolerance, `FunctionTolerance`, then the algorithm will stop. The solver will also stop when the objective function value reaches `ObjectiveLimit`. Finally the solver will stop after running for `MaxTime` seconds. Here we set the `FunctionTolerance` to 1e-5.

```
options.FunctionTolerance = 1e-5;
```

Run the `simulannealbnd` solver.

```
[x,fval,exitFlag,output] = simulannealbnd(ObjectiveFunction,startingPoint,lb,ub,options);
```

```
Optimization terminated: change in best function value less than options.FunctionTolerance.
```

```
fprintf('The number of iterations was : %d\n', output.iterations);
```

```
The number of iterations was : 1843
```

```
fprintf('The number of function evaluations was : %d\n', output.funccount);
```

```
The number of function evaluations was : 1864
```

```
fprintf('The best function value found was : %g\n', fval);
```

```
The best function value found was : 6.90334
```

How Simulated Annealing Works

Outline of the Algorithm

The simulated annealing algorithm performs the following steps:

1. The algorithm generates a random trial point. The algorithm chooses the distance of the trial point from the current point by a probability distribution with a scale depending on the current temperature. You set the trial point distance distribution as a function with the `AnnealingFcn` option. Choices:

   - `@annealingfast` (default) — Step length equals the current temperature, and direction is uniformly random.
   - `@annealingboltz` — Step length equals the square root of temperature, and direction is uniformly random.
   - `@myfun` — Custom annealing algorithm, `myfun`. For custom annealing function syntax, see Algorithm Settings.

   After generating the trial point, the algorithm shifts it, if necessary, to stay within bounds. The algorithm shifts each infeasible component of the trial point to a value chosen uniformly at random between the violated bound and the (feasible) value at the previous iteration.

2. The algorithm determines whether the new point is better or worse than the current point. If the new point is better than the current point, it becomes the next point. If the new point is worse than the current point, the algorithm can still make it the next point. The algorithm accepts a worse point based on an acceptance function. Choose the acceptance function with the `AcceptanceFcn` option. Choices:

   - `@acceptancesa` (default) — Simulated annealing acceptance function. The probability of acceptance is

     $$\frac{1}{1+\exp\left(\frac{\Delta}{\max(T)}\right)},$$

     where

     Δ = new objective – old objective.
     T_0 = initial temperature of component i
     T = the current temperature.

     Since both Δ and T are positive, the probability of acceptance is between 0 and 1/2. Smaller temperature leads to smaller acceptance probability. Also, larger Δ leads to smaller acceptance probability.

   - `@myfun` — Custom acceptance function, `myfun`. For custom acceptance function syntax, see Algorithm Settings.

3. The algorithm systematically lowers the temperature, storing the best point found so far. The `TemperatureFcn` option specifies the function the algorithm uses to update the temperature. Let k denote the annealing parameter. (The annealing parameter is the same as the iteration number until reannealing.) Options:

   - `@temperatureexp` (default) — $T = T_0 * 0.95\text{^}k$.

- `@temperaturefast` — $T = T_0 / k$.
- `@temperatureboltz` — $T = T_0 / \log(k)$.
- `@myfun` — Custom temperature function, `myfun`. For custom temperature function syntax, see Temperature Options.

4. `simulannealbnd` reanneals after it accepts `ReannealInterval` points. Reannealing sets the annealing parameters to lower values than the iteration number, thus raising the temperature in each dimension. The annealing parameters depend on the values of estimated gradients of the objective function in each dimension. The basic formula is

$$k_i = \log\left(\frac{T_0}{T_i}\frac{\max_j(s_j)}{s_i}\right),$$

where

k_i = annealing parameter for component i.
T_0 = initial temperature of component i.
T_i = current temperature of component i.
s_i = gradient of objective in direction i times difference of bounds in direction i.

`simulannealbnd` safeguards the annealing parameter values against `Inf` and other improper values.

5. The algorithm stops when the average change in the objective function is small relative to `FunctionTolerance`, or when it reaches any other stopping criterion. See Stopping Conditions for the Algorithm.

For more information on the algorithm, see Ingber [1].

Stopping Conditions for the Algorithm

The simulated annealing algorithm uses the following conditions to determine when to stop:

- `FunctionTolerance` — The algorithm runs until the average change in value of the objective function in `StallIterLim` iterations is less than the value of `FunctionTolerance`. The default value is `1e-6`.
- `MaxIterations` — The algorithm stops when the number of iterations exceeds this maximum number of iterations. You can specify the maximum number of iterations as a positive integer or `Inf`. The default value is `Inf`.
- `MaxFunctionEvaluations` specifies the maximum number of evaluations of the objective function. The algorithm stops if the number of function evaluations exceeds the value of `MaxFunctionEvaluations`. The default value is `3000*numberofvariables`.
- `MaxTime` specifies the maximum time in seconds the algorithm runs before stopping. The default value is `Inf`.
- `ObjectiveLimit` — The algorithm stops when the best objective function value is less than the value of `ObjectiveLimit`. The default value is `-Inf`.

Bibliography

[1] Ingber, L. *Adaptive simulated annealing (ASA): Lessons learned*. Invited paper to a special issue of the *Polish Journal Control and Cybernetics* on "Simulated Annealing Applied to Combinatorial Optimization." 1995. Available from `https://www.ingber.com/asa96_lessons.ps.gz`

See Also

`simulannealbnd`

Related Topics

- What Is Simulated Annealing?
- Simulated Annealing Terminology
- Minimize Function with Many Local Minima
- Minimization Using Simulated Annealing Algorithm