Documentation for minimize.h and minimize.c

Steven Andrews, © 2007-2023

History

3/2007	Initial parts of library written as part of DoRA project, while at MSI.
10/2008	Refurbished.
4/2009	Added `mnmz_setparamptr`.
2016	Additional work, still for DoRA project.
7/2023	Library added to Smoldyn files but not actually used.

Overview

The minimize library is a collection of routines for finding the minimum value of a function. The library is typically used by first calling `mnmz_alloc` to create a data structure for the minimization process, then registering the parameters that are to be optimized with `mnmz_setparam`. Then, call one of the minimization functions, each of which takes only one step, until a suitable result has been achieved. At the end, free the data structure. None of the search functions are particularly good, and some may contain bugs.

Header file

```
#ifndef __minimize_h
#define __minimize_h

typedef struct minimizestruct {
  int maxparam;                // maximum number of parameters
  int nparam;                  // actual number of parameters
  double **param;              // pointers to parameter variables
  double *priorparam;          // values of parameters prior to fitting
  double *scale;               // characteristic scale for each parameter
  double *lo;                  // minimum value for each parameter
  double *hi;                  // maximum value for each parameter
  int *fix;                    // 1 if parameter is fixed, 0 if not
  void *systemptr;             // pointer to system, which is passed to minfn
  double (*minfn)(void *,double);// function that is to be minimized
  double distance;             // the most recently achieved best distance
  double f1,f2,f3;             // floats for minimization functions
  double *v1,*v2,*v3;          // vectors for minimization functions
  double *m1;                  // matrix for minimization function
  } *minimizeptr;


minimizeptr mnmz_alloc(int maxparam,void* systemptr,double
(*minfn)(void*,double));
void mnmz_free(minimizeptr mnmz);
void mnmz_clear(minimizeptr mnmz);
int mnmz_setparamptr(minimizeptr mnmz,char *param,void *value);
int mnmz_setparam(minimizeptr mnmz,double *paramptr,double scale,double
```

```
lo,double hi,int fix);
int mnmz_step1(minimizeptr mnmz,int rptstep);
int mnmz_step2(minimizeptr mnmz,int rptstep);
int mnmz_step3(minimizeptr mnmz,int rptstep);
int mnmz_annealstep(minimizeptr mnmz,int rptstep);
void mnmz_randstep(minimizeptr mnmz,double change);
int mnmz_simplex(minimizeptr mnmz,int rptstep);

# endif
```

Data structure

This structure contains information about the function to be minimized, the parameters of that function, and some minimization routine variables. The function called `minfn` is the function to be minimized. Its return value is called distance, although that may not be the best name for it. The items that are sent to the `minfn` function are the `systemptr`, which is completely untouched by minimization routines, and the old distance. If the current distance is greater than the old distance, the `minfn` is supposed to stop calculating and just return whatever was found because it will be ignored anyhow. This method is used to allow efficient program execution.

In the structure, `param`, `priorparam`, `scale`, `lo`, `hi`, and `fix` each have `maxparam` elements. The vectors `v1`, `v2`, and `v3` each have `maxparam+1` elements and `m1` has size $(maxparam+1)^2$ elements. The scalars `f1`, `f2`, and `f3`, the vectors `v1`, `v2`, and `v3`, and matrix `m1` are for the exclusive use of the minimization function.

The value of scale is roughly the expected deviation for a parameter. In other words, before minimization, each parameter value should be set to the best guess, and `scale` should give a sense of the expected search range for that particular parameter. The actual search range may end up being much larger or much smaller.

Structure handling functions

```
minimizeptr mnmz_alloc(int maxparam,void* systemptr,double
    (*minfn)(void*,double));
```

Allocates a minimize structure for a total size of `maxparam`, allocates all internal arrays, and sets all values to defaults. The `systemptr` and `minfn` pointer inputs are simply copied into the data structure. Returns `NULL` if space could not be allocated. Default values:

maxparam	maxparam
nparam	0
param[i]	NULL
priorparam[i]	0
scale[i]	1
lo[i]	DBL_MIN
hi[i]	DBL_MAX
fix[i]	1
distance	DBL_MAX

```
f1, f2, f3              0
v1[i], v2[i], v3[i]     0
m1[i*(maxparam+1)+j]    0
```

`void mnmz_free(minimizeptr mnmz);`
Frees a minimize structure. This does not free the pointers in the `param` list, nor the `systemptr`, because those are not owned by the minimize structure.

`void mnmz_clear(minimizeptr mnmz);`
Clears a minimize structure, but does not free it. `maxparam`, `systemptr`, and `minfn` are untouched. All other elements are set to the defaults that are listed above in the `mnmz_alloc` discussion.

```
int mnmz_setparamptr(minimizeptr mnmz,char *param,void *value);
```
Sets one of the pointers in the minimize structure. This sets the `systemptr` element to `value` if `param` is "systemptr" and the `minfn` element to `value` if `param` is "minfn". This returns 0 if `param` is one of these and 1 if `param` is anything else.

```
int mnmz_setparam(minimizeptr mnmz,double *paramptr,double scale,double
lo,double hi,int fix);
```
Either adds a new parameter to a minimize structure, or changes the minimization values for a parameter that is already in existence. Note that this does not change the value of the parameter. Send in `paramptr` pointing to the parameter, `scale` as the scaling value for the minimizing function, `lo` and `hi` as the fitting domain limits, and `fix` as the fixing value. Returns 1 for successful operation and 0 if there is no space in the structure to add this new parameter.

Minimzation functions

Several functions are given here. The `mnmz_step1` one is the least sophisticated, but is the one that I use most. I'm not sure if the others are intrinsically worse or if I haven't figured out the best parameterization for them yet.

```
int mnmz_step1(minimizeptr mnmz,int rptstep);
```
Does one step of a random search to minimize the function. Send in `rptstep` with 0 for the first step, which initializes the range and distance, and non-zero for subsequent steps. Returns 1 if the trial solution was an improvement, in which case the new parameters are stored, and 0 if the trial solution was worse, in which case the parameters are unchanged. If this function is run for several thousand successes, one generally gets reasonably close to the overall minimum. This function should work reasonably well, but is extremely crude and inefficient.

This works with a greedy random walk method. For a step, each free parameter is changed using an approximately Gaussian density with mean of the current value and standard deviation of `range*scale[i]`, where `i` is the parameter number; these trial values are reflected into the range between `lo` and `hi` as needed. If the new result is an improvement, then the parameters are kept and the range is increased by

10%; if it is worse, the parameters are reset and the range is decreased by 0.1%. Note that there is a single range value for all parameters, meaning that the search region is always proportional to `scale[i]` for each parameter.

```
int mnmz_step2(minimizeptr mnmz,int rptstep);
```
This is essentially identical to `mnmz_step1`, but uses a slightly different algorithm. This varies only a single randomly chosen parameter each time it is called, and it varies it by an amount that is proportional to `sigma[i]`, where `sigma[i]` is set to `scale[i]` at the first call and then increases by 20% for each improvement and decreases by 1% for each failure.

```
int mnmz_step3(minimizeptr mnmz,int rptstep);
```
This is essentially identical to `mnmz_step1`. If the trial solution is worse, then it returns, just as before. However, if the trial solution is an improvement, then it searches the vector in parameter space that points in the direction of the step to see what step length optimizes the result. This optimization again uses a greedy random walk method. This looks good, but I didn't proofread it carefully.

```
int mnmz_annealstep(minimizeptr mnmz,int rptstep);
```
This is essentially identical to `mnmz_step1`. It differs in that moves that increase distance are permitted with probability exp(-Δ*distance*/`kt`), where `kt` is a thermal energy. The value of `kt` is initialized to the initial distance value and decreases by 1% at each function call.

```
void mnmz_randstep(minimizeptr mnmz,double change);
```
Does one random step on the non-fixed parameters, with rms step length equal to `change` times the `scale` value for the respective parameter. The `lo` and `hi` bounds are still observed. This function does not consider any distance function at all, but only moves the parameters randomly. This can be useful for restarting the minimization procedure from a new starting point.

```
int mnmz_simplex(minimizeptr mnmz,int rptstep);
```
This uses a simplex procedure for minimization. I think that it's copied nearly verbatim from *Numerical Recipies in C*, but I have not checked it at all.

Internal Function

```
double smplxmove(minimizeptr mnmz,int ihi,double fac);
```
This is part of the simplex optimization function.