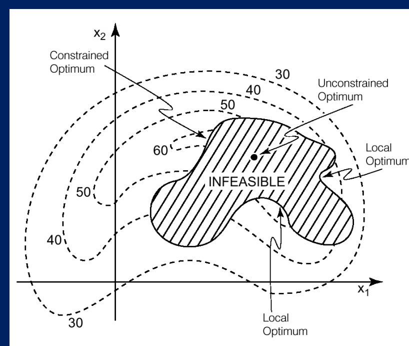
A New Method For Numerical Constrained Optimization



Ronald N. Perry
Mitsubishi Electric Research Laboratories

Motivation

- The applicability of optimization methods is widespread, reaching into almost *every* activity in which numerical information is processed
- For a summary of applications and theory
  - See Fletcher "Practical Methods of Optimization"
- For numerous applications in computer graphics
  - See Goldsmith and Barr "Applying constrained optimization to computer graphics"
- In this sketch, we describe a method and not its application

MERL

Informal Problem Statement

- An ideal problem for constrained optimization
  - has a single measure defining the quality of a solution (called the *objective function* F)
  - plus some requirements upon that solution that must not be violated (called the *constraints* C_i)
- A constrained optimization method maximizes (or minimizes) F while satisfying the C_i's
- Both F and C_i's are functions of $\mathbf{x} \in R^N$, the input parameters to be determined

MERL

Informal Problem Statement

- Many flavors of optimization
  - **x** can be real-valued, integer, mixed
  - F and C_i's can be linear, quadratic, nonlinear
  - F and C_i's can be smooth (i.e., differentiable) or nonsmooth
  - F and C_i's can be noisy or noise-free
  - methods can be globally convergent or global
- Our focus
  - globally convergent methods
  - real-valued, nonlinear, potentially nonsmooth, potentially noisy, constrained problems

MERL

Our Contribution

- A new method for constraint handling, called *partitioned performances*, that
  - can be applied to established optimization algorithms
  - can improve their ability to traverse constrained space
- A new optimization method, called *SPIDER*, that
  - applies partitioned performances to a new variation of the Nelder and Mead polytope algorithm

MERL

An observation leads to an idea

- Observation
  - Many constrained problems have optima that lie near constraint boundaries
  - Consequently, avoidance (or approximations) of constraints can hinder an algorithm's path to the answer
- Idea
  - By allowing (and even *encouraging*) an optimization algorithm to move its vertices into constrained space, a more efficient and robust algorithm emerges

MERL

The idea leads to a method

- Constraints are partitioned (i.e., grouped) into multiple levels (i.e., categories)
- A constrained performance, independent of the objective function, is defined for each level
- A set of rules, based on these *partitioned performances*, specify the ordering and movement of vertices as they straddle constraint boundaries
- These rules are non-greedy, permitting vertices at a higher (i.e., better) level to move to a lower (i.e., worse) level

MERL

Partitioned Performances (Advantages)

- Do not use a penalty function and thus do not warp the performance surface
  - this avoids the possible ill-conditioning of the objective function typical in penalty methods
- Do not linearize the constraints as do other methods (e.g., SQP)
- Assume very little about the problem form
  - F and Ci's can be nonsmooth (i.e., nondifferentiable) and highly nonlinear

MERL

Partitioning Constraints

- One effective partitioning of constraints
  - place simple limits on $\mathbf{x} \in R^N$ into level 1 (e.g., $x_1 \geq 0$)
  - place constraints which, when violated, produce singularities in F into level 1
  - all other constraints into level 2
  - and the objective function F into level 3
- Many different strategies for partitioning
  - just two levels: constrained and feasible
  - a level for every constraint, and a feasible level
  - *dynamic* partitioning (changing the level assignments during the search)

MERL

Computing Performance

- Assume a partitioning of F and the C_i's into W levels $[L_1 \ldots L_w]$ with $L_w = \{ F \}$
- We define the *partitioned performance* of a location $\mathbf{x} \in R^N$ as a 2-tuple <P,L> consisting of a floating point scalar P and an integer level indicator L. P represents the "goodness" of **x** at level L.

MERL

Computing Performance

- To determine <P,L>
  - sum the constraint violations in each level
  - L is assigned to the first level, beginning at level 1, to have any violation and P is assigned the sum of the violations at L
  - if no violations occur, $L \leftarrow W$ and $P \leftarrow F(\mathbf{x})$

MERL

Comparing Performances

- The partitioned performances of two locations $\mathbf{x_1}$ ($<P_1,L_1>$) and $\mathbf{x_2}$ ($<P_2,L_2>$) are compared as follows:
  - if ($L_1 == L_2$)
    - if ($P_1 > P_2$) $\mathbf{x_1}$ is better, otherwise $\mathbf{x_2}$ is better
  - if ($L_1 > L_2$)
    - $\mathbf{x_1}$ is better
  - if ($L_2 > L_1$)
    - $\mathbf{x_2}$ is better

MERL

SPIDER Method

- Applies partitioned performances to a new variation of the Nelder and Mead polytope algorithm
- Rules for ordering and movement using partitioned performances are demonstrated

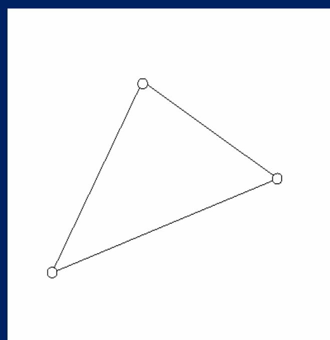
MERL

What is a "SPIDER"?

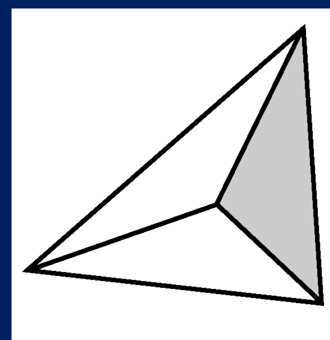
- Assuming we are maximizing an n-dimensional objective function F, SPIDER consists of n+1 "legs", where
  - each leg contains its position in space
  - associated with each leg is a partitioned performance

MERL

What is a “SPIDER”?



When n = 2, a triangle

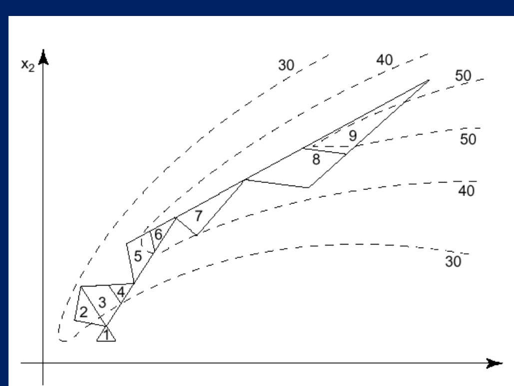


When n = 3, a tetrahedron

MERL

What does SPIDER do?

- Crawl: each leg is at a known “elevation” on the performance “hill”, and it is SPIDER’s task to crawl up the hill and maximize performance



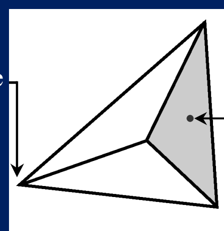
MERL

How SPIDER walks

- By moving each leg through the centroid of the remaining legs

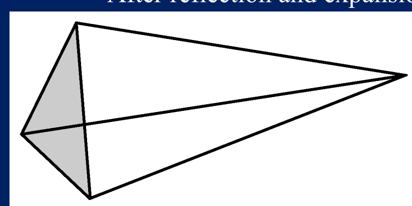
Before reflection and expansion

Leg to be moved



Centroid

After reflection and expansion



MERL

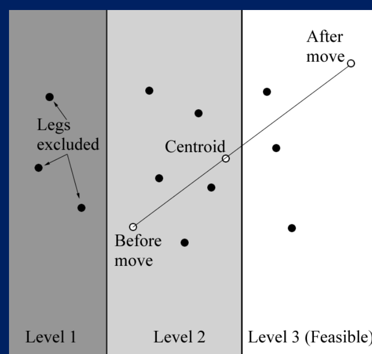
How SPIDER walks

- Repeat N times
  - Sort legs of SPIDER, from worst to best. Label worst and best legs.
  - For each leg L, in worst to best order
    - Determine centroid
    - Compute position and performance of a trial leg, L_{trial}
      - if L is not the best leg, reflect and expand *through* centroid
      - if L is the best leg, reflect and expand *away* from centroid
    - If move successful, accept trial, relabel worst and best leg if required
  - EndFor
  - Shrink SPIDER if best leg has not improved
  - Rebuild SPIDER if successive shrinks exceed threshold
- EndRepeat

MERL

Rules for centroid computation

- Exclude leg being moved (L)
- Exclude legs at a lower level than L
  - this helps to give SPIDER a better sense of direction along constraint boundaries



MERL

Rules for moving a non-best leg

- Same level (level of L_{trial} == level of L)
  - accept trial leg if
    - P value of L_{trial} > P value of L
- Going down levels (level of L_{trial} < level of L)
  - accept trial leg if its better than the worst leg
- Going up levels (level of L_{trial} > level of L)
  - accept trial leg if its better than the best leg

MERL

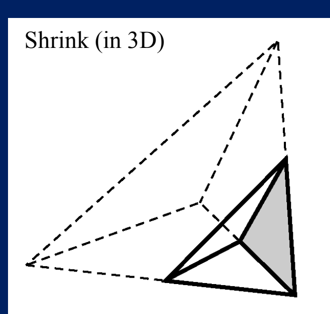
Rules for moving the best leg

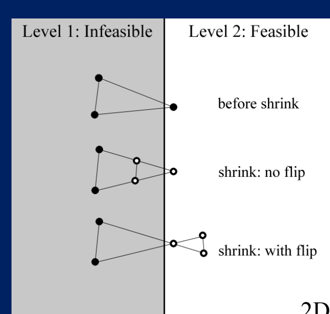
- It must improve in performance in order to move
- This gives SPIDER the ability to “straddle” and thus track along a constraint boundary

MERL

Rules for shrinking SPIDER

- Shrink the vertices at the same level as the best leg toward the best leg, and flip (as well as shrink) vertices at lower levels over the best leg
- Flipping helps to move legs across a constraint boundary towards feasibility





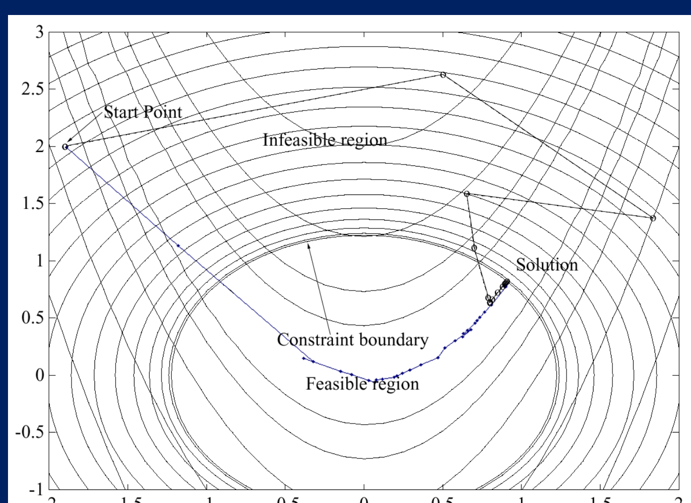
MERL

A Matlab Test Problem

- Sequential Quadratic Programming (SQP) methods represent the state-of-the-art in nonlinear constrained optimization
- SQP methods out perform every other tested method in terms of efficiency, accuracy, and percentage of successful solutions, over a large number of test problems
- On a Matlab test problem
  - Matlab SQP Implementation, 96 function calls
  - SPIDER, 108 function calls

MERL

A Matlab Test Problem



SPIDER walk in blue, SQP walk in black

MERL

The End

```
#include <stdio.h> #include <string.h> #define MAX_DIMS 20 #define MAX_LEGS 25 #define MAX_CONS 20 #define MAX_LEVELS 20 #define DFT_SHRINK_FACTOR 0.5f #define DFT_EXP_FACTOR
1.5f #define DFT_SHRINK_REB 7 #define DFT_BUILD_MFACTOR 0.0f #define DFT_LOG_PATH 0 #define DFT_VERBOSE 1 typedef float (*optObjFunc) (float *p); typedef float (*optConsFunc)
(float *p); typedef struct { int level; float perf; } Perf; typedef struct { int dim; optObjFunc objFunc; int numLevels; int numConsInLevel[MAX_LEVELS]; optConsFunc
consFunc[MAX_LEVELS] [MAX_CONS]; int numLegs; float legs[MAX_LEGS][MAX_DIMS]; Perf perf[MAX_LEGS]; int order[MAX_LEGS]; int bestLeg; int worstLeg; int curLeg; float
startPt[MAX_DIMS]; float sizeOfSpace[MAX_DIMS]; float centroid[MAX_DIMS]; float trial[MAX_DIMS]; float expFactor; float shrinkFactor; int numShrinksBeforeRebuild; int
numShrinks; float buildMultFactor; int logPath; int verbose; int numFuncCalls; } SPIDER; float Rosenbrocks (float *p) { float x1 = p[0]; float x2 = p[1]; float t1 = (x2 - x1 *
x1); float t2 = (1.0f - x1); float fx = 100.0f * t1 * t1 + t2 * t2; return(-fx); } float EllipCons (float *p) { float x1 = p[0]; float x2 = p[1]; float cx = x1 * x1 + x2 * x2 -
1.5f; return(-cx); } void SPIDERCycle (SPIDER *S, int cycleCount); void SPIDERBuild (SPIDER *S); void SPIDERScore (SPIDER *S); Perf SPIDERLegPerf (SPIDER *S, float *leg); int
SPIDERPerfBetter (Perf P1, Perf P2); int SPIDERPerfWorse (Perf P1, Perf P2); void SPIDERSetWorstLeg (SPIDER *S); void SPIDERCentroid (SPIDER *S); void SPIDERSort (SPIDER *S);
void SPIDERShrink (SPIDER *S); void SPIDERRebuild (SPIDER *S); void SPIDERPrint (SPIDER *S); void SPIDERCycle (SPIDER *S, int cycleCount) { Perf PTrial; Perf PCurrent; Perf
PBest; Perf PWorst; int i, j, n; int bestLegBetter; FILE *fd = 0; if (S->logPath) { fd = fopen("log.txt", "wt"); for (j = 0; j < S->dim; ++j) fprintf(fd, "%.4f ", S-
>startPt[j]); fprintf(fd, "\n"); } S->numShrinks = 0; for (n = 0; n < cycleCount; ++n) { bestLegBetter = 0; SPIDERSort(S); for (i = 0; i < S->numLegs; ++i) { S->curLeg = S-
>order[i]; SPIDERCentroid(S); if (S->curLeg == S->bestLeg) { for (j = 0; j < S->dim; ++j) { float span = S->centroid[j] - S->legs[S->curLeg][j]; S->trial[j] = S->centroid[j] -
span * S->expFactor; } PBest = S->perf[S->bestLeg]; PTrial = SPIDERLegPerf(S, S->trial); if (SPIDERPerfBetter(PTrial, PBest)) { for (j = 0; j < S->dim; ++j) S->legs[S-
>curLeg][j] = S->trial[j]; S->perf[S->curLeg] = PTrial; bestLegBetter = 1; continue; } } else { for (j = 0; j < S->dim; ++j) { float span = S->centroid[j] - S->legs[S-
>curLeg][j]; S->trial[j] = S->centroid[j] + span * S->expFactor; } PCurrent = S->perf[S->curLeg]; PBest = S->perf[S->bestLeg]; PWorst = S->perf[S->worstLeg]; PTrial =
SPIDERLegPerf(S, S->trial); if ((PTrial.level == PCurrent.level && PTrial.perf > PCurrent.perf) || (PTrial.level > PCurrent.level && SPIDERPerfBetter (PTrial, PBest)) ||
(PTrial.level < PCurrent.level && SPIDERPerfBetter(PTrial, PWorst))) { for (j = 0; j < S->dim; ++j) S->legs[S->curLeg][j] = S->trial[j]; S->perf[S->curLeg] = PTrial;
SPIDERSetWorstLeg(S); if (SPIDERPerfBetter(PTrial, PBest)) { S->bestLeg = S->curLeg; bestLegBetter = 1; } } } } if (!bestLegBetter) { if (S->numShrinks < S-
>numShrinksBeforeRebuild) { if (S->verbose) printf("Cycle: %.2d < ----- Shrink Required >\n", n + 1); SPIDERShrink(S); ++S->numShrinks; } else { if (S->verbose) printf("Cycle:
%.2d < ----- Rebuild Required >\n", n + 1); SPIDERRebuild(S); S->numShrinks = 0; } } else { S->numShrinks = 0; } if (S->verbose) { int legIdx = S->bestLeg; printf("Cycle: %.2d
", n + 1); printf("< FuncCalls %d, Level %d, Perf %.4f: ", S->numFuncCalls, S->perf[legIdx].level, S->perf[legIdx].perf); for (j = 0; j < S->dim; ++j) printf("%.4f ", S-
>legs[legIdx][j]); printf(">\n"); } if (S->logPath) { int legIdx = S->bestLeg; for (j = 0; j < S->dim; ++j) fprintf(fd, "%.4f ", S->legs[legIdx][j]); fprintf(fd, "\n"); } } if
(S->logPath) fclose(fd); SPIDERSort(S); } void SPIDERBuild (SPIDER *S) { int useRand = (S->buildMultFactor == 0.0f ? 1 : 0); if (S->numLegs == S->dim + 1) { int i, j; for (i =
0; i < S->numLegs; ++i) { for (j = 0; j < S->dim; ++j) { S->legs[i][j] = S->startPt[j]; if (i == j) { float r = (useRand ? (((float) rand()) / ((float) RAND_MAX)) : S-
>buildMultFactor); S->legs[i][j] += r * S->sizeOfSpace[j]; } } } } else { int i, j; for (i = 0; i < S->numLegs; ++i) { for (j = 0; j < S->dim; ++j) { S->legs[i][j] = S-
>startPt[j]; if (i != 0) { float r = (useRand ? (((float) rand()) / ((float) RAND_MAX)) : S->buildMultFactor); S->legs[i][j] += r * S->sizeOfSpace[j]; } } } } SPIDERScore(S);
SPIDERSort(S); } void SPIDERScore (SPIDER *S) { int n; for (n = 0; n < S->numLegs; ++n) S->perf[n] = SPIDERLegPerf(S, &S->legs[n][0]); } Perf SPIDERLegPerf (SPIDER *S, float
*leg) { ++S->numFuncCalls; if (S->numLevels == 0) { Perf P; P.level = 0; P.perf = (*S->objFunc)(leg); return(P); } else { Perf P; int i, n; for (n = 0; n < S->numLevels; ++n) {
float levelSum = 0.0f; for (i = 0; i < S->numConsInLevel[n]; ++i) { float consVal = (*S->consFunc[n][i])(leg); if (consVal < 0.0f) levelSum += consVal; } if (levelSum < 0.0f) {
P.level = n; P.perf = levelSum; return(P); } } P.level = S->numLevels; P.perf = (*S->objFunc)(leg); return(P); } } int SPIDERPerfBetter (Perf P1, Perf P2) { if (P1.level >
P2.level) return(1); if (P1.level < P2.level) return(0); if (P1.perf > P2.perf) return(1); return(0); } int SPIDERPerfWorse (Perf P1, Perf P2) { if (P1.level < P2.level)
return(1); if (P1.level > P2.level) return(0); if (P1.perf < P2.perf) return(1); return(0); } void SPIDERSetWorstLeg (SPIDER *S) { int n; Perf worstLegPerf = S->perf[0]; S-
>worstLeg = 0; for (n = 1; n < S->numLegs; ++n) { if (SPIDERPerfWorse(S->perf[n], worstLegPerf)) { S->worstLeg = n; worstLegPerf = S->perf[n]; } } } void SPIDERCentroid (SPIDER
*S) { int i, n; int numValidLegs = 0; int numCentroidLegs = 0; int curLeg = S->curLeg; for (i = 0; i < S->dim; ++i) S->centroid[i] = 0.0f; for (n = 0; n < S->numLegs; ++n) { if
(n == curLeg) continue; if (S->perf[n].level < S->perf[curLeg].level) continue; ++numValidLegs; } if (numValidLegs <= (S->numLegs / 2)) { for (n = 0; n < S->numLegs; ++n) { if
(n == curLeg) continue; for (i = 0; i < S->dim; ++i) S->centroid[i] += S->legs[n][i]; ++numCentroidLegs; } } else { for (n = 0; n < S->numLegs; ++n) { if (n == curLeg)
continue; if (S->perf[n].level < S->perf[curLeg].level) continue; for (i = 0; i < S->dim; ++i) S->centroid[i] += S->legs[n][i]; ++numCentroidLegs; } } for (i = 0; i < S->dim;
++i) S->centroid[i] /= (float) numCentroidLegs; } void SPIDERSort (SPIDER *S) { int i, j; for (i = 0; i < S->numLegs; i++) S->order[i] = i; for (i = 0; i < S->numLegs; i++) {
for (j = 1; j < (S->numLegs - i); j++) { Perf P1 = S->perf[S->order[j - 1]]; Perf P2 = S->perf[S->order[j]]; if (P1.level > P2.level || (P1.level == P2.level && P1.perf >
P2.perf)) { int t; t = S->order[j - 1]; S->order[j - 1] = S->order[j]; S->order[j] = t; } } } S->worstLeg = S->order[0]; S->bestLeg = S->order[S->numLegs - 1]; } void
SPIDERShrink (SPIDER *S) { int i, j; int bestLeg = S->bestLeg; int bestLegLevel = S->perf[bestLeg].level; float shrinkFactor = S->shrinkFactor; for (i = 0; i < S->numLegs; ++i)
{ if (i == bestLeg) continue; if (S->perf[i].level == bestLegLevel) { for (j = 0; j < S->dim; ++j) { S->legs[i][j] = shrinkFactor * S->legs[i][j] + (1.0f - shrinkFactor) * S-
>legs[bestLeg][j]; } } else { for (j = 0; j < S->dim; ++j) { float coord = S->legs[i][j]; S->legs[i][j] = coord + (S->legs[bestLeg][j] - coord) * (1.0f + shrinkFactor); } } }
SPIDERScore(S); } void SPIDERRebuild (SPIDER *S) { int n; float startPt[MAX_DIMS]; for (n = 0; n < S->dim; ++n) { startPt[n] = S->startPt[n]; S->startPt[n] = S->legs[S-
>bestLeg][n]; } SPIDERBuild(S); for (n = 0; n < S->dim; ++n) S->startPt[n] = startPt[n]; } void SPIDERPrint (SPIDER *S) { int i, j; printf("Func calls: %d\n", S->numFuncCalls);
printf("Exp Factor: %.2f\n", S->expFactor); printf("Srk Factor: %.2f\n", S->shrinkFactor); printf("Bld Factor: %.2f\n", S->buildMultFactor); SPIDERSort(S); for (i = 0; i < S-
>numLegs; ++i) { int legIdx = S->order[i]; printf("Leg %d, Level %d, Perf %.4f: ", legIdx + 1, S->perf[legIdx].level, S->perf[legIdx].perf); for (j = 0; j < S->dim; ++j)
printf("%.4f ", S->legs[legIdx][j]); printf("\n"); } } void main (void) { SPIDER S; S.dim = 2; S.objFunc = Rosenbrocks; S.numLevels = 1; S.numConsInLevel[0] = 1;
S.consFunc[0][0] = EllipCons; S.numLegs = 3; S.startPt[0] = -1.9f; S.startPt[1] = +2.0f; S.sizeOfSpace[0] = 1.0f; S.sizeOfSpace[1] = 1.0f; S.expFactor = DFT_EXP_FACTOR;
S.shrinkFactor = DFT_SHRINK_FACTOR; S.numShrinksBeforeRebuild = DFT_SHRINK_REB; S.buildMultFactor = DFT_BUILD_MFACTOR; S.logPath = DFT_LOG_PATH; S.verbose = DFT_VERBOSE;
S.numFuncCalls = 0; SPIDERBuild(&S); SPIDERCycle(&S, 50); }
```

MERL