

```
30  * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
31  * ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
32  * POSSIBILITY OF SUCH DAMAGE.
33  *****/
34  //
35  // Navigation function computation
36  // Uses Dijkstra's method
37  // Modified for Euclidean-distance computation
38  //
39  // Path calculation uses no interpolation when pot field is at max in
40  //   nearby cells
41  //
42  // Path calc has sanity check that it succeeded
43  //
44
45
46  #include <navfn/navfn.h>
47  #include <ros/console.h>
48
49  namespace navfn {
50
51      //
52      // function to perform nav fn calculation
53      // keeps track of internal buffers, will be more efficient
54      //   if the size of the environment does not change
55      //
56
57      int
58      create_nav_plan_astar(COSTTYPE *costmap, int nx, int ny,
59                          int* goal, int* start,
60                          float *plan, int nplan)
61      {
62          static NavFn *nav = NULL;
63
64          if (nav == NULL)
65              nav = new NavFn(nx,ny);
66
67          if (nav->nx != nx || nav->ny != ny) // check for compatibility with previous call
68          {
69              delete nav;
70              nav = new NavFn(nx,ny);
71          }
72
73          nav->setGoal(goal);
74          nav->setStart(start);
75
76          nav->costarr = costmap;
77          nav->setupNavFn(true);
78
```

```
79 // calculate the nav fn and path
80 nav->priInc = 2*COST_NEUTRAL;
81 nav->propNavFnAstar(std::max(nx*ny/20, nx+ny));
82
83 // path
84 int len = nav->calcPath(nplan);
85
86 if (len > 0) // found plan
87     ROS_DEBUG("[NavFn] Path found, %d steps\n", len);
88 else
89     ROS_DEBUG("[NavFn] No path found\n");
90
91 if (len > 0)
92 {
93     for (int i=0; i<len; i++)
94     {
95         plan[i*2] = nav->pathx[i];
96         plan[i*2+1] = nav->pathy[i];
97     }
98 }
99
100 return len;
101 }
102
103
104
105
106 //
107 // create nav fn buffers
108 //
109
110 NavFn::NavFn(int xs, int ys)
111 {
112     // create cell arrays
113     costarr = NULL;
114     potarr = NULL;
115     pending = NULL;
116     gradx = grady = NULL;
117     setNavArr(xs,ys);
118
119     // priority buffers
120     pb1 = new int[PRIORITYBUFSIZE];
121     pb2 = new int[PRIORITYBUFSIZE];
122     pb3 = new int[PRIORITYBUFSIZE];
123
124     // for Dijkstra (breadth-first), set to COST_NEUTRAL
125     // for A* (best-first), set to COST_NEUTRAL
126     priInc = 2*COST_NEUTRAL;
127
```

```
128     // goal and start
129     goal[0] = goal[1] = 0;
130     start[0] = start[1] = 0;
131
132     // display function
133     displayFn = NULL;
134     displayInt = 0;
135
136     // path buffers
137     npathbuf = npath = 0;
138     pathx = pathy = NULL;
139     pathStep = 0.5;
140 }
141
142
143 NavFn::~NavFn()
144 {
145     if(costarr)
146         delete[] costarr;
147     if(potarr)
148         delete[] potarr;
149     if(pending)
150         delete[] pending;
151     if(gradx)
152         delete[] gradx;
153     if(grady)
154         delete[] grady;
155     if(pathx)
156         delete[] pathx;
157     if(pathy)
158         delete[] pathy;
159     if(pb1)
160         delete[] pb1;
161     if(pb2)
162         delete[] pb2;
163     if(pb3)
164         delete[] pb3;
165 }
166
167
168 //
169 // set goal, start positions for the nav fn
170 //
171
172 void
173 NavFn::setGoal(int *g)
174 {
175     goal[0] = g[0];
176     goal[1] = g[1];
```

```
177     ROS_DEBUG("[NavFn] Setting goal to %d,%d\n", goal[0], goal[1]);
178 }
179
180 void
181 NavFn::setStart(int *g)
182 {
183     start[0] = g[0];
184     start[1] = g[1];
185     ROS_DEBUG("[NavFn] Setting start to %d,%d\n", start[0], start[1]);
186 }
187
188 //
189 // Set/Reset map size
190 //
191
192 void
193 NavFn::setNavArr(int xs, int ys)
194 {
195     ROS_DEBUG("[NavFn] Array is %d x %d\n", xs, ys);
196
197     nx = xs;
198     ny = ys;
199     ns = nx*ny;
200
201     if(costarr)
202         delete[] costarr;
203     if(potarr)
204         delete[] potarr;
205     if(pending)
206         delete[] pending;
207
208     if(gradx)
209         delete[] gradx;
210     if(grady)
211         delete[] grady;
212
213     costarr = new COSTTYPE[ns]; // cost array, 2d config space
214     memset(costarr, 0, ns*sizeof(COSTTYPE));
215     potarr = new float[ns]; // navigation potential array
216     pending = new bool[ns];
217     memset(pending, 0, ns*sizeof(bool));
218     gradx = new float[ns];
219     grady = new float[ns];
220 }
221
222
223 //
224 // set up cost array, usually from ROS
225 //
```

```
226
227 void
228 NavFn::setCostmap(const COSTTYPE *cmap, bool isROS, bool allow_unknown)
229 {
230     COSTTYPE *cm = costarr;
231     if (isROS) // ROS-type cost array
232     {
233         for (int i=0; i<ny; i++)
234         {
235             int k=i*nx;
236             for (int j=0; j<nx; j++, k++, cmap++, cm++)
237             {
238                 // This transforms the incoming cost values:
239                 // COST_OBS -> COST_OBS (incoming "lethal obstacle")
240                 // COST_OBS_ROS -> COST_OBS (incoming "inscribed inflated obstacle")
241                 // values in range 0 to 252 -> values from COST_NEUTRAL to COST_OBS_ROS.
242                 *cm = COST_OBS;
243                 int v = *cmap;
244                 if (v < COST_OBS_ROS)
245                 {
246                     v = COST_NEUTRAL+COST_FACTOR*v;
247                     if (v >= COST_OBS)
248                         v = COST_OBS-1;
249                     *cm = v;
250                 }
251                 else if(v == COST_UNKNOWN_ROS && allow_unknown)
252                 {
253                     v = COST_OBS-1;
254                     *cm = v;
255                 }
256             }
257         }
258     }
259
260     else // not a ROS map, just a PGM
261     {
262         for (int i=0; i<ny; i++)
263         {
264             int k=i*nx;
265             for (int j=0; j<nx; j++, k++, cmap++, cm++)
266             {
267                 *cm = COST_OBS;
268                 if (i<7 || i > ny-8 || j<7 || j > nx-8)
269                     continue; // don't do borders
270                 int v = *cmap;
271                 if (v < COST_OBS_ROS)
272                 {
273                     v = COST_NEUTRAL+COST_FACTOR*v;
274                     if (v >= COST_OBS)
```

```
275         v = COST_OBS-1;
276         *cm = v;
277     }
278     else if(v == COST_UNKNOWN_ROS)
279     {
280         v = COST_OBS-1;
281         *cm = v;
282     }
283 }
284 }
285
286 }
287 }
288
289 bool
290 NavFn::calcNavFnDijkstra(bool atStart)
291 {
292     setupNavFn(true);
293
294     // calculate the nav fn and path
295     propNavFnDijkstra(std::max(nx*ny/20,nx+ny),atStart);
296
297     // path
298     int len = calcPath(nx*ny/2);
299
300     if (len > 0) // found plan
301     {
302         ROS_DEBUG("[NavFn] Path found, %d steps\n", len);
303         return true;
304     }
305     else
306     {
307         ROS_DEBUG("[NavFn] No path found\n");
308         return false;
309     }
310
311 }
312
313
314 //
315 // calculate navigation function, given a costmap, goal, and start
316 //
317
318 bool
319 NavFn::calcNavFnAstar()
320 {
321     setupNavFn(true);
322
323     // calculate the nav fn and path
```

```
324     propNavFnAstar(std::max(nx*ny/20, nx+ny));
325
326     // path
327     int len = calcPath(nx*4);
328
329     if (len > 0) // found plan
330     {
331         ROS_DEBUG("[NavFn] Path found, %d steps\n", len);
332         return true;
333     }
334     else
335     {
336         ROS_DEBUG("[NavFn] No path found\n");
337         return false;
338     }
339 }
340
341
342 //
343 // returning values
344 //
345
346 float *NavFn::getPathX() { return pathx; }
347 float *NavFn::getPathY() { return pathy; }
348 int    NavFn::getPathLen() { return npath; }
349
350 // inserting onto the priority blocks
351 #define push_cur(n) { if (n>=0 && n<ns && !pending[n] && \
352     costarr[n]<COST_OBS && curPe<PRIORITYBUFSIZE) \
353     { curP[curPe++]=n; pending[n]=true; }}
354 #define push_next(n) { if (n>=0 && n<ns && !pending[n] && \
355     costarr[n]<COST_OBS && nextPe<PRIORITYBUFSIZE) \
356     { nextP[nextPe++]=n; pending[n]=true; }}
357 #define push_over(n) { if (n>=0 && n<ns && !pending[n] && \
358     costarr[n]<COST_OBS && overPe<PRIORITYBUFSIZE) \
359     { overP[overPe++]=n; pending[n]=true; }}
360
361
362 // Set up navigation potential arrays for new propagation
363
364 void
365 NavFn::setupNavFn(bool keepit)
366 {
367     // reset values in propagation arrays
368     for (int i=0; i<ns; i++)
369     {
370         potarr[i] = POT_HIGH;
371         if (!keepit) costarr[i] = COST_NEUTRAL;
372         gradx[i] = grady[i] = 0.0;
```

```
373     }
374
375     // outer bounds of cost array
376     COSTTYPE *pc;
377     pc = costarr;
378     for (int i=0; i<nx; i++)
379         *pc++ = COST_OBS;
380     pc = costarr + (ny-1)*nx;
381     for (int i=0; i<nx; i++)
382         *pc++ = COST_OBS;
383     pc = costarr;
384     for (int i=0; i<ny; i++, pc+=nx)
385         *pc = COST_OBS;
386     pc = costarr + nx - 1;
387     for (int i=0; i<ny; i++, pc+=nx)
388         *pc = COST_OBS;
389
390     // priority buffers
391     curT = COST_OBS;
392     curP = pb1;
393     curPe = 0;
394     nextP = pb2;
395     nextPe = 0;
396     overP = pb3;
397     overPe = 0;
398     memset(pending, 0, ns*sizeof(bool));
399
400     // set goal
401     int k = goal[0] + goal[1]*nx;
402     initCost(k,0);
403
404     // find # of obstacle cells
405     pc = costarr;
406     int ntot = 0;
407     for (int i=0; i<ns; i++, pc++)
408     {
409         if (*pc >= COST_OBS)
410             ntot++; // number of cells that are obstacles
411     }
412     nobs = ntot;
413 }
414
415
416 // initialize a goal-type cost for starting propagation
417
418 void
419 NavFn::initCost(int k, float v)
420 {
421     potarr[k] = v;
```



```
422     push_cur(k+1);
423     push_cur(k-1);
424     push_cur(k-nx);
425     push_cur(k+nx);
426 }
427
428
429 //
430 // Critical function: calculate updated potential value of a cell,
431 //   given its neighbors' values
432 // Planar-wave update calculation from two lowest neighbors in a 4-grid
433 // Quadratic approximation to the interpolated value
434 // No checking of bounds here, this function should be fast
435 //
436
437 #define INVSQRT2 0.707106781
438
439 inline void
440 NavFn::updateCell(int n)
441 {
442     // get neighbors
443     float u,d,l,r;
444     l = potarr[n-1];
445     r = potarr[n+1];
446     u = potarr[n-nx];
447     d = potarr[n+nx];
448     // ROS_INFO("Update] c: %0.1f  l: %0.1f  r: %0.1f  u: %0.1f  d: %0.1f\n",
449     //           potarr[n], l, r, u, d);
450     // ROS_INFO("Update] cost: %d\n", costarr[n]);
451
452     // find lowest, and its lowest neighbor
453     float ta, tc;
454     if (l<r) tc=l; else tc=r;
455     if (u<d) ta=u; else ta=d;
456
457     // do planar wave update
458     if (costarr[n] < COST_OBS) // don't propagate into obstacles
459     {
460         float hf = (float)costarr[n]; // traversability factor
461         float dc = tc-ta; // relative cost between ta,tc
462         if (dc < 0) // ta is lowest
463         {
464             dc = -dc;
465             ta = tc;
466         }
467
468         // calculate new potential
469         float pot;
470         if (dc >= hf) // if too large, use ta-only update
```

```
471     pot = ta+hf;
472 else                                     // two-neighbor interpolation update
473 {
474     // use quadratic approximation
475     // might speed this up through table lookup, but still have to
476     // do the divide
477     float d = dc/hf;
478     float v = -0.2301*d*d + 0.5307*d + 0.7040;
479     pot = ta + hf*v;
480 }
481
482 //     ROS_INFO("[Update] new pot: %d\n", costarr[n]);
483
484 // now add affected neighbors to priority blocks
485 if (pot < potarr[n])
486 {
487     float le = INVSQRT2*(float)costarr[n-1];
488     float re = INVSQRT2*(float)costarr[n+1];
489     float ue = INVSQRT2*(float)costarr[n-nx];
490     float de = INVSQRT2*(float)costarr[n+nx];
491     potarr[n] = pot;
492     if (pot < curT)           // low-cost buffer block
493     {
494         if (l > pot+le) push_next(n-1);
495         if (r > pot+re) push_next(n+1);
496         if (u > pot+ue) push_next(n-nx);
497         if (d > pot+de) push_next(n+nx);
498     }
499     else                     // overflow block
500     {
501         if (l > pot+le) push_over(n-1);
502         if (r > pot+re) push_over(n+1);
503         if (u > pot+ue) push_over(n-nx);
504         if (d > pot+de) push_over(n+nx);
505     }
506 }
507
508 }
509
510 }
511
512
513 //
514 // Use A* method for setting priorities
515 // Critical function: calculate updated potential value of a cell,
516 // given its neighbors' values
517 // Planar-wave update calculation from two lowest neighbors in a 4-grid
518 // Quadratic approximation to the interpolated value
519 // No checking of bounds here, this function should be fast
```

```
520 //
521
522 #define INVSQRT2 0.707106781
523
524 inline void
525 NavFn::updateCellAstar(int n)
526 {
527     // get neighbors
528     float u,d,l,r;
529     l = potarr[n-1];
530     r = potarr[n+1];
531     u = potarr[n-nx];
532     d = potarr[n+nx];
533     //ROS_INFO("[Update] c: %0.1f l: %0.1f r: %0.1f u: %0.1f d: %0.1f\n",
534     //          potarr[n], l, r, u, d);
535     // ROS_INFO("[Update] cost of %d: %d\n", n, costarr[n]);
536
537     // find lowest, and its lowest neighbor
538     float ta, tc;
539     if (l<r) tc=l; else tc=r;
540     if (u<d) ta=u; else ta=d;
541
542     // do planar wave update
543     if (costarr[n] < COST_OBS) // don't propagate into obstacles
544     {
545         float hf = (float)costarr[n]; // traversability factor
546         float dc = tc-ta; // relative cost between ta,tc
547         if (dc < 0) // ta is lowest
548         {
549             dc = -dc;
550             ta = tc;
551         }
552
553         // calculate new potential
554         float pot;
555         if (dc >= hf) // if too large, use ta-only update
556             pot = ta+hf;
557         else // two-neighbor interpolation update
558         {
559             // use quadratic approximation
560             // might speed this up through table lookup, but still have to
561             // do the divide
562             float d = dc/hf;
563             float v = -0.2301*d*d + 0.5307*d + 0.7040;
564             pot = ta + hf*v;
565         }
566
567         //ROS_INFO("[Update] new pot: %d\n", costarr[n]);
568
```

```
569 // now add affected neighbors to priority blocks
570 if (pot < potarr[n])
571 {
572     float le = INVSQRT2*(float)costarr[n-1];
573     float re = INVSQRT2*(float)costarr[n+1];
574     float ue = INVSQRT2*(float)costarr[n-nx];
575     float de = INVSQRT2*(float)costarr[n+nx];
576
577     // calculate distance
578     int x = n%nx;
579     int y = n/nx;
580     float dist = hypot(x-start[0], y-start[1])*(float)COST_NEUTRAL;
581
582     potarr[n] = pot;
583     pot += dist;
584     if (pot < curT) // low-cost buffer block
585     {
586         if (l > pot+le) push_next(n-1);
587         if (r > pot+re) push_next(n+1);
588         if (u > pot+ue) push_next(n-nx);
589         if (d > pot+de) push_next(n+nx);
590     }
591     else
592     {
593         if (l > pot+le) push_over(n-1);
594         if (r > pot+re) push_over(n+1);
595         if (u > pot+ue) push_over(n-nx);
596         if (d > pot+de) push_over(n+nx);
597     }
598 }
599
600 }
601
602 }
603
604
605
606 //
607 // main propagation function
608 // Dijkstra method, breadth-first
609 // runs for a specified number of cycles,
610 // or until it runs out of cells to update,
611 // or until the Start cell is found (atStart = true)
612 //
613
614 bool
615 NavFn::propNavFnDijkstra(int cycles, bool atStart)
616 {
617     int nwx = 0; // max priority block size
```

```
618     int nc = 0; // number of cells put into priority blocks
619     int cycle = 0; // which cycle we're on
620
621     // set up start cell
622     int startCell = start[1]*nx + start[0];
623
624     for (; cycle < cycles; cycle++) // go for this many cycles, unless interrupted
625     {
626         //
627         if (curPe == 0 && nextPe == 0) // priority blocks empty
628             break;
629
630         // stats
631         nc += curPe;
632         if (curPe > nwv)
633             nwv = curPe;
634
635         // reset pending flags on current priority buffer
636         int *pb = curP;
637         int i = curPe;
638         while (i-- > 0)
639             pending[*pb++] = false;
640
641         // process current priority buffer
642         pb = curP;
643         i = curPe;
644         while (i-- > 0)
645             updateCell(*pb++);
646
647         if (displayInt > 0 && (cycle % displayInt) == 0)
648             displayFn(this);
649
650         // swap priority blocks curP <=> nextP
651         curPe = nextPe;
652         nextPe = 0;
653         pb = curP; // swap buffers
654         curP = nextP;
655         nextP = pb;
656
657         // see if we're done with this priority level
658         if (curPe == 0)
659         {
660             curT += priInc; // increment priority threshold
661             curPe = overPe; // set current to overflow block
662             overPe = 0;
663             pb = curP; // swap buffers
664             curP = overP;
665             overP = pb;
666         }
```

```
667
668     // check if we've hit the Start cell
669     if (atStart)
670         if (potarr[startCell] < POT_HIGH)
671             break;
672 }
673
674 ROS_DEBUG("[NavFn] Used %d cycles, %d cells visited (%d%%), priority buf max %d\n",
675           cycle, nc, (int)((nc*100.0)/(ns-nobs)), nwv);
676
677 if (cycle < cycles) return true; // finished up here
678 else return false;
679 }
680
681
682 //
683 // main propagation function
684 // A* method, best-first
685 // uses Euclidean distance heuristic
686 // runs for a specified number of cycles,
687 // or until it runs out of cells to update,
688 // or until the Start cell is found (atStart = true)
689 //
690
691 bool
692 NavFn::propNavFnAstar(int cycles)
693 {
694     int nwv = 0; // max priority block size
695     int nc = 0; // number of cells put into priority blocks
696     int cycle = 0; // which cycle we're on
697
698     // set initial threshold, based on distance
699     float dist = hypot(goal[0]-start[0], goal[1]-start[1])*(float)COST_NEUTRAL;
700     curT = dist + curT;
701
702     // set up start cell
703     int startCell = start[1]*nx + start[0];
704
705     // do main cycle
706     for (; cycle < cycles; cycle++) // go for this many cycles, unless interrupted
707     {
708         //
709         if (curPe == 0 && nextPe == 0) // priority blocks empty
710             break;
711
712         // stats
713         nc += curPe;
714         if (curPe > nwv)
715             nwv = curPe;
```

```
716
717     // reset pending flags on current priority buffer
718     int *pb = curP;
719     int i = curPe;
720     while (i-- > 0)
721         pending[(pb++)] = false;
722
723     // process current priority buffer
724     pb = curP;
725     i = curPe;
726     while (i-- > 0)
727         updateCellAstar(*pb++);
728
729     if (displayInt > 0 && (cycle % displayInt) == 0)
730         displayFn(this);
731
732     // swap priority blocks curP <=> nextP
733     curPe = nextPe;
734     nextPe = 0;
735     pb = curP;           // swap buffers
736     curP = nextP;
737     nextP = pb;
738
739     // see if we're done with this priority level
740     if (curPe == 0)
741     {
742         curT += priInc;      // increment priority threshold
743         curPe = overPe;     // set current to overflow block
744         overPe = 0;
745         pb = curP;          // swap buffers
746         curP = overP;
747         overP = pb;
748     }
749
750     // check if we've hit the Start cell
751     if (potarr[startCell] < POT_HIGH)
752         break;
753
754 }
755
756 last_path_cost_ = potarr[startCell];
757
758 ROS_DEBUG("[NavFn] Used %d cycles, %d cells visited (%d%%), priority buf max %d\n",
759           cycle, nc, (int)((nc*100.0)/(ns-nobs)), nwv);
760
761
762 if (potarr[startCell] < POT_HIGH) return true; // finished up here
763 else return false;
764 }
```

```
765
766
767     float NavFn::getLastPathCost()
768     {
769         return last_path_cost_;
770     }
771
772
773     //
774     // Path construction
775     // Find gradient at array points, interpolate path
776     // Use step size of pathStep, usually 0.5 pixel
777     //
778     // Some sanity checks:
779     // 1. Stuck at same index position
780     // 2. Doesn't get near goal
781     // 3. Surrounded by high potentials
782     //
783
784     int
785     NavFn::calcPath(int n, int *st)
786     {
787         // test write
788         //savemap("test");
789
790         // check path arrays
791         if (npathbuf < n)
792         {
793             if (pathx) delete [] pathx;
794             if (pathy) delete [] pathy;
795             pathx = new float[n];
796             pathy = new float[n];
797             npathbuf = n;
798         }
799
800         // set up start position at cell
801         // st is always upper left corner for 4-point bilinear interpolation
802         if (st == NULL) st = start;
803         int stc = st[1]*nx + st[0];
804
805         // set up offset
806         float dx=0;
807         float dy=0;
808         npath = 0;
809
810         // go for <n> cycles at most
811         for (int i=0; i<n; i++)
812         {
813             // check if near goal
```



```

814     int nearest_point=std::max(0, std::min(nx*ny-1, stc+(int)round(dx)+(int)(nx*round(dy
815     if (potarr[nearest_point] < COST_NEUTRAL)
816     {
817         pathx[npath] = (float)goal[0];
818         pathy[npath] = (float)goal[1];
819         return ++npath;        // done!
820     }
821
822     if (stc < nx || stc > ns-nx) // would be out of bounds
823     {
824         ROS_DEBUG("[PathCalc] Out of bounds");
825         return 0;
826     }
827
828     // add to path
829     pathx[npath] = stc%nx + dx;
830     pathy[npath] = stc/nx + dy;
831     npath++;
832
833     bool oscillation_detected = false;
834     if( npath > 2 &&
835         pathx[npath-1] == pathx[npath-3] &&
836         pathy[npath-1] == pathy[npath-3] )
837     {
838         ROS_DEBUG("[PathCalc] oscillation detected, attempting fix.");
839         oscillation_detected = true;
840     }
841
842     int stcnx = stc+nx;
843     int stcpv = stc-nx;
844
845     // check for potentials at eight positions near cell
846     if (potarr[stc] >= POT_HIGH ||
847         potarr[stc+1] >= POT_HIGH ||
848         potarr[stc-1] >= POT_HIGH ||
849         potarr[stcnx] >= POT_HIGH ||
850         potarr[stcnx+1] >= POT_HIGH ||
851         potarr[stcnx-1] >= POT_HIGH ||
852         potarr[stcpv] >= POT_HIGH ||
853         potarr[stcpv+1] >= POT_HIGH ||
854         potarr[stcpv-1] >= POT_HIGH ||
855         oscillation_detected)
856     {
857         ROS_DEBUG("[Path] Pot fn boundary, following grid (%0.1f/%d)", potarr[stc], npat
858         // check eight neighbors to find the lowest
859         int minc = stc;
860         int minp = potarr[stc];
861         int st = stcpv - 1;
862         if (potarr[st] < minp) {minp = potarr[st]; minc = st; }

```

```
863     st++;
864     if (potarr[st] < minp) {minp = potarr[st]; minc = st; }
865     st++;
866     if (potarr[st] < minp) {minp = potarr[st]; minc = st; }
867     st = stc-1;
868     if (potarr[st] < minp) {minp = potarr[st]; minc = st; }
869     st = stc+1;
870     if (potarr[st] < minp) {minp = potarr[st]; minc = st; }
871     st = stcnx-1;
872     if (potarr[st] < minp) {minp = potarr[st]; minc = st; }
873     st++;
874     if (potarr[st] < minp) {minp = potarr[st]; minc = st; }
875     st++;
876     if (potarr[st] < minp) {minp = potarr[st]; minc = st; }
877     stc = minc;
878     dx = 0;
879     dy = 0;
880
881     ROS_DEBUG("[Path] Pot: %0.1f  pos: %0.1f,%0.1f",
882               potarr[stc], pathx[npath-1], pathy[npath-1]);
883
884     if (potarr[stc] >= POT_HIGH)
885     {
886         ROS_DEBUG("[PathCalc] No path found, high potential");
887         //savemap("navfn_highpot");
888         return 0;
889     }
890 }
891
892 // have a good gradient here
893 else
894 {
895
896     // get grad at four positions near cell
897     gradCell(stc);
898     gradCell(stc+1);
899     gradCell(stcnx);
900     gradCell(stcnx+1);
901
902
903     // get interpolated gradient
904     float x1 = (1.0-dx)*gradx[stc] + dx*gradx[stc+1];
905     float x2 = (1.0-dx)*gradx[stcnx] + dx*gradx[stcnx+1];
906     float x = (1.0-dy)*x1 + dy*x2; // interpolated x
907     float y1 = (1.0-dx)*grady[stc] + dx*grady[stc+1];
908     float y2 = (1.0-dx)*grady[stcnx] + dx*grady[stcnx+1];
909     float y = (1.0-dy)*y1 + dy*y2; // interpolated y
910
911     // show gradients
```

```
912     ROS_DEBUG("[Path] %0.2f,%0.2f %0.2f,%0.2f %0.2f,%0.2f %0.2f,%0.2f; final x=%.",
913             gradx[stc], grady[stc], gradx[stc+1], grady[stc+1],
914             gradx[stcnx], grady[stcnx], gradx[stcnx+1], grady[stcnx+1],
915             x, y);
916
917     // check for zero gradient, failed
918     if (x == 0.0 && y == 0.0)
919     {
920         ROS_DEBUG("[PathCalc] Zero gradient");
921         return 0;
922     }
923
924     // move in the right direction
925     float ss = pathStep/hypot(x, y);
926     dx += x*ss;
927     dy += y*ss;
928
929     // check for overflow
930     if (dx > 1.0) { stc++; dx -= 1.0; }
931     if (dx < -1.0) { stc--; dx += 1.0; }
932     if (dy > 1.0) { stc+=nx; dy -= 1.0; }
933     if (dy < -1.0) { stc-=nx; dy += 1.0; }
934
935 }
936
937 //      ROS_INFO("[Path] Pot: %0.1f grad: %0.1f,%0.1f pos: %0.1f,%0.1f\n",
938 //      potarr[stc], x, y, pathx[npath-1], pathy[npath-1]);
939 }
940
941 // return npath; // out of cycles, return failure
942 ROS_DEBUG("[PathCalc] No path found, path too long");
943 //savemap("navfn_pathlong");
944 return 0; // out of cycles, return failure
945 }
946
947
948 //
949 // gradient calculations
950 //
951
952 // calculate gradient at a cell
953 // positive value are to the right and down
954 float
955 NavFn::gradCell(int n)
956 {
957     if (gradx[n]+grady[n] > 0.0) // check this cell
958         return 1.0;
959
960     if (n < nx || n > ns-nx) // would be out of bounds
```

```
961         return 0.0;
962
963     float cv = potarr[n];
964     float dx = 0.0;
965     float dy = 0.0;
966
967     // check for in an obstacle
968     if (cv >= POT_HIGH)
969     {
970         if (potarr[n-1] < POT_HIGH)
971             dx = -COST_OBS;
972         else if (potarr[n+1] < POT_HIGH)
973             dx = COST_OBS;
974
975         if (potarr[n-nx] < POT_HIGH)
976             dy = -COST_OBS;
977         else if (potarr[n+nx] < POT_HIGH)
978             dy = COST_OBS;
979     }
980
981     else // not in an obstacle
982     {
983         // dx calc, average to sides
984         if (potarr[n-1] < POT_HIGH)
985             dx += potarr[n-1] - cv;
986         if (potarr[n+1] < POT_HIGH)
987             dx += cv - potarr[n+1];
988
989         // dy calc, average to sides
990         if (potarr[n-nx] < POT_HIGH)
991             dy += potarr[n-nx] - cv;
992         if (potarr[n+nx] < POT_HIGH)
993             dy += cv - potarr[n+nx];
994     }
995
996     // normalize
997     float norm = hypot(dx, dy);
998     if (norm > 0)
999     {
1000         norm = 1.0/norm;
1001         gradx[n] = norm*dx;
1002         grady[n] = norm*dy;
1003     }
1004     return norm;
1005 }
1006
1007
1008 //
1009 // display function setup
```

```
1010 // <n> is the number of cycles to wait before displaying,
1011 //     use 0 to turn it off
1012
1013 void
1014 NavFn::display(void fn(NavFn *nav), int n)
1015 {
1016     displayFn = fn;
1017     displayInt = n;
1018 }
1019
1020
1021 //
1022 // debug writes
1023 // saves costmap and start/goal
1024 //
1025
1026 void
1027 NavFn::savemap(const char *fname)
1028 {
1029     char fn[4096];
1030
1031     ROS_DEBUG("[NavFn] Saving costmap and start/goal points");
1032     // write start and goal points
1033     sprintf(fn, "%s.txt", fname);
1034     FILE *fp = fopen(fn, "w");
1035     if (!fp)
1036     {
1037         ROS_WARN("Can't open file %s", fn);
1038         return;
1039     }
1040     fprintf(fp, "Goal: %d %d\nStart: %d %d\n", goal[0], goal[1], start[0], start[1]);
1041     fclose(fp);
1042
1043     // write cost array
1044     if (!costarr) return;
1045     sprintf(fn, "%s.pgm", fname);
1046     fp = fopen(fn, "wb");
1047     if (!fp)
1048     {
1049         ROS_WARN("Can't open file %s", fn);
1050         return;
1051     }
1052     fprintf(fp, "P5\n%d\n%d\n%d\n", nx, ny, 0xff);
1053     fwrite(costarr, 1, nx*ny, fp);
1054     fclose(fp);
1055 }
1056 };
```