```cpp
*  LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
*  ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
*  POSSIBILITY OF SUCH DAMAGE.
*********************************************************************/
//
// Navigation function computation
// Uses Dijkstra's method
// Modified for Euclidean-distance computation
//
// Path calculation uses no interpolation when pot field is at max in
//   nearby cells
//
// Path calc has sanity check that it succeeded
//


#include <navfn/navfn.h>
#include <ros/console.h>

namespace navfn {

  //
  // function to perform nav fn calculation
  // keeps track of internal buffers, will be more efficient
  //   if the size of the environment does not change
  //

  int
    create_nav_plan_astar(COSTTYPE *costmap, int nx, int ny,
        int* goal, int* start,
        float *plan, int nplan)
    {
      static NavFn *nav = NULL;

      if (nav == NULL)
        nav = new NavFn(nx,ny);

      if (nav->nx != nx || nav->ny != ny) // check for compatibility with previous call
      {
        delete nav;
        nav = new NavFn(nx,ny);
      }

      nav->setGoal(goal);
      nav->setStart(start);

      nav->costarr = costmap;
      nav->setupNavFn(true);

```

```
    // calculate the nav fn and path
    nav->priInc = 2*COST_NEUTRAL;
    nav->propNavFnAstar(std::max(nx*ny/20,nx+ny));

    // path
    int len = nav->calcPath(nplan);

    if (len > 0)                      // found plan
      ROS_DEBUG("[NavFn] Path found, %d steps\n", len);
    else
      ROS_DEBUG("[NavFn] No path found\n");

    if (len > 0)
    {
      for (int i=0; i<len; i++)
      {
        plan[i*2] = nav->pathx[i];
        plan[i*2+1] = nav->pathy[i];
      }
    }

    return len;
  }




//
// create nav fn buffers
//

NavFn::NavFn(int xs, int ys)
{
  // create cell arrays
  costarr = NULL;
  potarr = NULL;
  pending = NULL;
  gradx = grady = NULL;
  setNavArr(xs,ys);

  // priority buffers
  pb1 = new int[PRIORITYBUFSIZE];
  pb2 = new int[PRIORITYBUFSIZE];
  pb3 = new int[PRIORITYBUFSIZE];

  // for Dijkstra (breadth-first), set to COST_NEUTRAL
  // for A* (best-first), set to COST_NEUTRAL
  priInc = 2*COST_NEUTRAL;

```

```
    // goal and start
    goal[0] = goal[1] = 0;
    start[0] = start[1] = 0;

    // display function
    displayFn = NULL;
    displayInt = 0;

    // path buffers
    npathbuf = npath = 0;
    pathx = pathy = NULL;
    pathStep = 0.5;
  }


  NavFn::~NavFn()
  {
    if(costarr)
      delete[] costarr;
    if(potarr)
      delete[] potarr;
    if(pending)
      delete[] pending;
    if(gradx)
      delete[] gradx;
    if(grady)
      delete[] grady;
    if(pathx)
      delete[] pathx;
    if(pathy)
      delete[] pathy;
    if(pb1)
      delete[] pb1;
    if(pb2)
      delete[] pb2;
    if(pb3)
      delete[] pb3;
  }


  //
  // set goal, start positions for the nav fn
  //

  void
    NavFn::setGoal(int *g)
    {
      goal[0] = g[0];
      goal[1] = g[1];
```

```
      ROS_DEBUG("[NavFn] Setting goal to %d,%d\n", goal[0], goal[1]);
    }

  void
    NavFn::setStart(int *g)
    {
      start[0] = g[0];
      start[1] = g[1];
      ROS_DEBUG("[NavFn] Setting start to %d,%d\n", start[0], start[1]);
    }

  //
  // Set/Reset map size
  //

  void
    NavFn::setNavArr(int xs, int ys)
    {
      ROS_DEBUG("[NavFn] Array is %d x %d\n", xs, ys);

      nx = xs;
      ny = ys;
      ns = nx*ny;

      if(costarr)
        delete[] costarr;
      if(potarr)
        delete[] potarr;
      if(pending)
        delete[] pending;

      if(gradx)
        delete[] gradx;
      if(grady)
        delete[] grady;

      costarr = new COSTTYPE[ns]; // cost array, 2d config space
      memset(costarr, 0, ns*sizeof(COSTTYPE));
      potarr = new float[ns];   // navigation potential array
      pending = new bool[ns];
      memset(pending, 0, ns*sizeof(bool));
      gradx = new float[ns];
      grady = new float[ns];
    }


  //
  // set up cost array, usually from ROS
  //
```

```
  void
    NavFn::setCostmap(const COSTTYPE *cmap, bool isROS, bool allow_unknown)
    {
      COSTTYPE *cm = costarr;
      if (isROS)                          // ROS-type cost array
      {
        for (int i=0; i<ny; i++)
        {
          int k=i*nx;
          for (int j=0; j<nx; j++, k++, cmap++, cm++)
          {
            // This transforms the incoming cost values:
            // COST_OBS                 -> COST_OBS (incoming "lethal obstacle")
            // COST_OBS_ROS             -> COST_OBS (incoming "inscribed inflated obstacle
            // values in range 0 to 252 -> values from COST_NEUTRAL to COST_OBS_ROS.
            *cm = COST_OBS;
            int v = *cmap;
            if (v < COST_OBS_ROS)
            {
              v = COST_NEUTRAL+COST_FACTOR*v;
              if (v >= COST_OBS)
                v = COST_OBS-1;
              *cm = v;
            }
            else if(v == COST_UNKNOWN_ROS && allow_unknown)
            {
              v = COST_OBS-1;
              *cm = v;
            }
          }
        }
      }

      else                                // not a ROS map, just a PGM
      {
        for (int i=0; i<ny; i++)
        {
          int k=i*nx;
          for (int j=0; j<nx; j++, k++, cmap++, cm++)
          {
            *cm = COST_OBS;
            if (i<7 || i > ny-8 || j<7 || j > nx-8)
              continue; // don't do borders
            int v = *cmap;
            if (v < COST_OBS_ROS)
            {
              v = COST_NEUTRAL+COST_FACTOR*v;
              if (v >= COST_OBS)
```

```
                v = COST_OBS-1;
              *cm = v;
            }
            else if(v == COST_UNKNOWN_ROS)
            {
              v = COST_OBS-1;
              *cm = v;
            }
          }
        }

      }
    }

  bool
    NavFn::calcNavFnDijkstra(bool atStart)
    {
      setupNavFn(true);

      // calculate the nav fn and path
      propNavFnDijkstra(std::max(nx*ny/20,nx+ny),atStart);

      // path
      int len = calcPath(nx*ny/2);

      if (len > 0)                      // found plan
      {
        ROS_DEBUG("[NavFn] Path found, %d steps\n", len);
        return true;
      }
      else
      {
        ROS_DEBUG("[NavFn] No path found\n");
        return false;
      }

    }


  //
  // calculate navigation function, given a costmap, goal, and start
  //

  bool
    NavFn::calcNavFnAstar()
    {
      setupNavFn(true);

      // calculate the nav fn and path
```

```
      propNavFnAstar(std::max(nx*ny/20,nx+ny));

      // path
      int len = calcPath(nx*4);

      if (len > 0)                      // found plan
      {
        ROS_DEBUG("[NavFn] Path found, %d steps\n", len);
        return true;
      }
      else
      {
        ROS_DEBUG("[NavFn] No path found\n");
        return false;
      }
    }


  //
  // returning values
  //

  float *NavFn::getPathX() { return pathx; }
  float *NavFn::getPathY() { return pathy; }
  int    NavFn::getPathLen() { return npath; }

  // inserting onto the priority blocks
#define push_cur(n)  { if (n>=0 && n<ns && !pending[n] && \
    costarr[n]<COST_OBS && curPe<PRIORITYBUFSIZE) \
  { curP[curPe++]=n; pending[n]=true; }}
#define push_next(n) { if (n>=0 && n<ns && !pending[n] && \
    costarr[n]<COST_OBS && nextPe<PRIORITYBUFSIZE) \
  { nextP[nextPe++]=n; pending[n]=true; }}
#define push_over(n) { if (n>=0 && n<ns && !pending[n] && \
    costarr[n]<COST_OBS && overPe<PRIORITYBUFSIZE) \
  { overP[overPe++]=n; pending[n]=true; }}


  // Set up navigation potential arrays for new propagation

  void
    NavFn::setupNavFn(bool keepit)
    {
      // reset values in propagation arrays
      for (int i=0; i<ns; i++)
      {
        potarr[i] = POT_HIGH;
        if (!keepit) costarr[i] = COST_NEUTRAL;
        gradx[i] = grady[i] = 0.0;
```

```
    }

    // outer bounds of cost array
    COSTTYPE *pc;
    pc = costarr;
    for (int i=0; i<nx; i++)
      *pc++ = COST_OBS;
    pc = costarr + (ny-1)*nx;
    for (int i=0; i<nx; i++)
      *pc++ = COST_OBS;
    pc = costarr;
    for (int i=0; i<ny; i++, pc+=nx)
      *pc = COST_OBS;
    pc = costarr + nx - 1;
    for (int i=0; i<ny; i++, pc+=nx)
      *pc = COST_OBS;

    // priority buffers
    curT = COST_OBS;
    curP = pb1;
    curPe = 0;
    nextP = pb2;
    nextPe = 0;
    overP = pb3;
    overPe = 0;
    memset(pending, 0, ns*sizeof(bool));

    // set goal
    int k = goal[0] + goal[1]*nx;
    initCost(k,0);

    // find # of obstacle cells
    pc = costarr;
    int ntot = 0;
    for (int i=0; i<ns; i++, pc++)
    {
      if (*pc >= COST_OBS)
        ntot++;                       // number of cells that are obstacles
    }
    nobs = ntot;
  }


// initialize a goal-type cost for starting propagation

void
  NavFn::initCost(int k, float v)
  {
    potarr[k] = v;
```

```
      push_cur(k+1);
      push_cur(k-1);
      push_cur(k-nx);
      push_cur(k+nx);
    }


  //
  // Critical function: calculate updated potential value of a cell,
  //   given its neighbors' values
  // Planar-wave update calculation from two lowest neighbors in a 4-grid
  // Quadratic approximation to the interpolated value
  // No checking of bounds here, this function should be fast
  //

#define INVSQRT2 0.707106781

  inline void
    NavFn::updateCell(int n)
    {
      // get neighbors
      float u,d,l,r;
      l = potarr[n-1];
      r = potarr[n+1];
      u = potarr[n-nx];
      d = potarr[n+nx];
      //  ROS_INFO("[Update] c: %0.1f  l: %0.1f  r: %0.1f  u: %0.1f  d: %0.1f\n",
      //         potarr[n], l, r, u, d);
      //  ROS_INFO("[Update] cost: %d\n", costarr[n]);

      // find lowest, and its lowest neighbor
      float ta, tc;
      if (l<r) tc=l; else tc=r;
      if (u<d) ta=u; else ta=d;

      // do planar wave update
      if (costarr[n] < COST_OBS)        // don't propagate into obstacles
      {
        float hf = (float)costarr[n]; // traversability factor
        float dc = tc-ta;               // relative cost between ta,tc
        if (dc < 0)             // ta is lowest
        {
          dc = -dc;
          ta = tc;
        }

        // calculate new potential
        float pot;
        if (dc >= hf)           // if too large, use ta-only update
```

```
        pot = ta+hf;
      else                    // two-neighbor interpolation update
      {
        // use quadratic approximation
        // might speed this up through table lookup, but still have to
        //   do the divide
        float d = dc/hf;
        float v = -0.2301*d*d + 0.5307*d + 0.7040;
        pot = ta + hf*v;
      }

      //      ROS_INFO("[Update] new pot: %d\n", costarr[n]);

      // now add affected neighbors to priority blocks
      if (pot < potarr[n])
      {
        float le = INVSQRT2*(float)costarr[n-1];
        float re = INVSQRT2*(float)costarr[n+1];
        float ue = INVSQRT2*(float)costarr[n-nx];
        float de = INVSQRT2*(float)costarr[n+nx];
        potarr[n] = pot;
        if (pot < curT)       // low-cost buffer block
        {
          if (l > pot+le) push_next(n-1);
          if (r > pot+re) push_next(n+1);
          if (u > pot+ue) push_next(n-nx);
          if (d > pot+de) push_next(n+nx);
        }
        else                  // overflow block
        {
          if (l > pot+le) push_over(n-1);
          if (r > pot+re) push_over(n+1);
          if (u > pot+ue) push_over(n-nx);
          if (d > pot+de) push_over(n+nx);
        }
      }

    }

  }


//
// Use A* method for setting priorities
// Critical function: calculate updated potential value of a cell,
//   given its neighbors' values
// Planar-wave update calculation from two lowest neighbors in a 4-grid
// Quadratic approximation to the interpolated value
// No checking of bounds here, this function should be fast
```

```
  //

#define INVSQRT2 0.707106781

  inline void
    NavFn::updateCellAstar(int n)
    {
      // get neighbors
      float u,d,l,r;
      l = potarr[n-1];
      r = potarr[n+1];
      u = potarr[n-nx];
      d = potarr[n+nx];
      //ROS_INFO("[Update] c: %0.1f  l: %0.1f  r: %0.1f  u: %0.1f  d: %0.1f\n",
      //         potarr[n], l, r, u, d);
      // ROS_INFO("[Update] cost of %d: %d\n", n, costarr[n]);

      // find lowest, and its lowest neighbor
      float ta, tc;
      if (l<r) tc=l; else tc=r;
      if (u<d) ta=u; else ta=d;

      // do planar wave update
      if (costarr[n] < COST_OBS)        // don't propagate into obstacles
      {
        float hf = (float)costarr[n]; // traversability factor
        float dc = tc-ta;               // relative cost between ta,tc
        if (dc < 0)             // ta is lowest
        {
          dc = -dc;
          ta = tc;
        }

        // calculate new potential
        float pot;
        if (dc >= hf)           // if too large, use ta-only update
          pot = ta+hf;
        else                    // two-neighbor interpolation update
        {
          // use quadratic approximation
          // might speed this up through table lookup, but still have to
          //   do the divide
          float d = dc/hf;
          float v = -0.2301*d*d + 0.5307*d + 0.7040;
          pot = ta + hf*v;
        }

        //ROS_INFO("[Update] new pot: %d\n", costarr[n]);

```

```
        // now add affected neighbors to priority blocks
        if (pot < potarr[n])
        {
          float le = INVSQRT2*(float)costarr[n-1];
          float re = INVSQRT2*(float)costarr[n+1];
          float ue = INVSQRT2*(float)costarr[n-nx];
          float de = INVSQRT2*(float)costarr[n+nx];

          // calculate distance
          int x = n%nx;
          int y = n/nx;
          float dist = hypot(x-start[0], y-start[1])*(float)COST_NEUTRAL;

          potarr[n] = pot;
          pot += dist;
          if (pot < curT)       // low-cost buffer block
          {
            if (l > pot+le) push_next(n-1);
            if (r > pot+re) push_next(n+1);
            if (u > pot+ue) push_next(n-nx);
            if (d > pot+de) push_next(n+nx);
          }
          else
          {
            if (l > pot+le) push_over(n-1);
            if (r > pot+re) push_over(n+1);
            if (u > pot+ue) push_over(n-nx);
            if (d > pot+de) push_over(n+nx);
          }
        }

      }

    }



  //
  // main propagation function
  // Dijkstra method, breadth-first
  // runs for a specified number of cycles,
  //   or until it runs out of cells to update,
  //   or until the Start cell is found (atStart = true)
  //

  bool
    NavFn::propNavFnDijkstra(int cycles, bool atStart)
    {
      int nwv = 0;                      // max priority block size
```

```
    int nc = 0;                       // number of cells put into priority blocks
    int cycle = 0;            // which cycle we're on

    // set up start cell
    int startCell = start[1]*nx + start[0];

    for (; cycle < cycles; cycle++) // go for this many cycles, unless interrupted
    {
      //
      if (curPe == 0 && nextPe == 0) // priority blocks empty
        break;

      // stats
      nc += curPe;
      if (curPe > nwv)
        nwv = curPe;

      // reset pending flags on current priority buffer
      int *pb = curP;
      int i = curPe;
      while (i-- > 0)
        pending[*(pb++)] = false;

      // process current priority buffer
      pb = curP;
      i = curPe;
      while (i-- > 0)
        updateCell(*pb++);

      if (displayInt > 0 &&  (cycle % displayInt) == 0)
        displayFn(this);

      // swap priority blocks curP <=> nextP
      curPe = nextPe;
      nextPe = 0;
      pb = curP;              // swap buffers
      curP = nextP;
      nextP = pb;

      // see if we're done with this priority level
      if (curPe == 0)
      {
        curT += priInc;       // increment priority threshold
        curPe = overPe;       // set current to overflow block
        overPe = 0;
        pb = curP;            // swap buffers
        curP = overP;
        overP = pb;
      }
```

```cpp

        // check if we've hit the Start cell
        if (atStart)
          if (potarr[startCell] < POT_HIGH)
            break;
      }

      ROS_DEBUG("[NavFn] Used %d cycles, %d cells visited (%d%%), priority buf max %d\n",
          cycle,nc,(int)((nc*100.0)/(ns-nobs)),nwv);

      if (cycle < cycles) return true; // finished up here
      else return false;
    }


  //
  // main propagation function
  // A* method, best-first
  // uses Euclidean distance heuristic
  // runs for a specified number of cycles,
  //   or until it runs out of cells to update,
  //   or until the Start cell is found (atStart = true)
  //

  bool
    NavFn::propNavFnAstar(int cycles)
    {
      int nwv = 0;                        // max priority block size
      int nc = 0;                         // number of cells put into priority blocks
      int cycle = 0;              // which cycle we're on

      // set initial threshold, based on distance
      float dist = hypot(goal[0]-start[0], goal[1]-start[1])*(float)COST_NEUTRAL;
      curT = dist + curT;

      // set up start cell
      int startCell = start[1]*nx + start[0];

      // do main cycle
      for (; cycle < cycles; cycle++) // go for this many cycles, unless interrupted
      {
        //
        if (curPe == 0 && nextPe == 0) // priority blocks empty
          break;

        // stats
        nc += curPe;
        if (curPe > nwv)
          nwv = curPe;
```

```

        // reset pending flags on current priority buffer
        int *pb = curP;
        int i = curPe;
        while (i-- > 0)
          pending[*(pb++)] = false;

        // process current priority buffer
        pb = curP;
        i = curPe;
        while (i-- > 0)
          updateCellAstar(*pb++);

        if (displayInt > 0 &&  (cycle % displayInt) == 0)
          displayFn(this);

        // swap priority blocks curP <=> nextP
        curPe = nextPe;
        nextPe = 0;
        pb = curP;              // swap buffers
        curP = nextP;
        nextP = pb;

        // see if we're done with this priority level
        if (curPe == 0)
        {
          curT += priInc;       // increment priority threshold
          curPe = overPe;       // set current to overflow block
          overPe = 0;
          pb = curP;            // swap buffers
          curP = overP;
          overP = pb;
        }

        // check if we've hit the Start cell
        if (potarr[startCell] < POT_HIGH)
          break;

      }

      last_path_cost_ = potarr[startCell];

      ROS_DEBUG("[NavFn] Used %d cycles, %d cells visited (%d%%), priority buf max %d\n",
          cycle,nc,(int)((nc*100.0)/(ns-nobs)),nwv);


      if (potarr[startCell] < POT_HIGH) return true; // finished up here
      else return false;
    }
```

```


float NavFn::getLastPathCost()
{
  return last_path_cost_;
}


//
// Path construction
// Find gradient at array points, interpolate path
// Use step size of pathStep, usually 0.5 pixel
//
// Some sanity checks:
//  1. Stuck at same index position
//  2. Doesn't get near goal
//  3. Surrounded by high potentials
//

int
  NavFn::calcPath(int n, int *st)
  {
    // test write
    //savemap("test");

    // check path arrays
    if (npathbuf < n)
    {
      if (pathx) delete [] pathx;
      if (pathy) delete [] pathy;
      pathx = new float[n];
      pathy = new float[n];
      npathbuf = n;
    }

    // set up start position at cell
    // st is always upper left corner for 4-point bilinear interpolation
    if (st == NULL) st = start;
    int stc = st[1]*nx + st[0];

    // set up offset
    float dx=0;
    float dy=0;
    npath = 0;

    // go for <n> cycles at most
    for (int i=0; i<n; i++)
    {
      // check if near goal
```

```
      int nearest_point=std::max(0,std::min(nx*ny-1,stc+(int)round(dx)+(int)(nx*round(dy
      if (potarr[nearest_point] < COST_NEUTRAL)
      {
        pathx[npath] = (float)goal[0];
        pathy[npath] = (float)goal[1];
        return ++npath;       // done!
      }

      if (stc < nx || stc > ns-nx) // would be out of bounds
      {
        ROS_DEBUG("[PathCalc] Out of bounds");
        return 0;
      }

      // add to path
      pathx[npath] = stc%nx + dx;
      pathy[npath] = stc/nx + dy;
      npath++;

      bool oscillation_detected = false;
      if( npath > 2 &&
          pathx[npath-1] == pathx[npath-3] &&
          pathy[npath-1] == pathy[npath-3] )
      {
        ROS_DEBUG("[PathCalc] oscillation detected, attempting fix.");
        oscillation_detected = true;
      }

      int stcnx = stc+nx;
      int stcpx = stc-nx;

      // check for potentials at eight positions near cell
      if (potarr[stc] >= POT_HIGH ||
          potarr[stc+1] >= POT_HIGH ||
          potarr[stc-1] >= POT_HIGH ||
          potarr[stcnx] >= POT_HIGH ||
          potarr[stcnx+1] >= POT_HIGH ||
          potarr[stcnx-1] >= POT_HIGH ||
          potarr[stcpx] >= POT_HIGH ||
          potarr[stcpx+1] >= POT_HIGH ||
          potarr[stcpx-1] >= POT_HIGH ||
          oscillation_detected)
      {
        ROS_DEBUG("[Path] Pot fn boundary, following grid (%0.1f/%d)", potarr[stc], npat
        // check eight neighbors to find the lowest
        int minc = stc;
        int minp = potarr[stc];
        int st = stcpx - 1;
        if (potarr[st] < minp) {minp = potarr[st]; minc = st; }
```

```
          st++;
          if (potarr[st] < minp) {minp = potarr[st]; minc = st; }
          st++;
          if (potarr[st] < minp) {minp = potarr[st]; minc = st; }
          st = stc-1;
          if (potarr[st] < minp) {minp = potarr[st]; minc = st; }
          st = stc+1;
          if (potarr[st] < minp) {minp = potarr[st]; minc = st; }
          st = stcnx-1;
          if (potarr[st] < minp) {minp = potarr[st]; minc = st; }
          st++;
          if (potarr[st] < minp) {minp = potarr[st]; minc = st; }
          st++;
          if (potarr[st] < minp) {minp = potarr[st]; minc = st; }
          stc = minc;
          dx = 0;
          dy = 0;

          ROS_DEBUG("[Path] Pot: %0.1f  pos: %0.1f,%0.1f",
              potarr[stc], pathx[npath-1], pathy[npath-1]);

          if (potarr[stc] >= POT_HIGH)
          {
            ROS_DEBUG("[PathCalc] No path found, high potential");
            //savemap("navfn_highpot");
            return 0;
          }
        }

        // have a good gradient here
        else
        {

          // get grad at four positions near cell
          gradCell(stc);
          gradCell(stc+1);
          gradCell(stcnx);
          gradCell(stcnx+1);


          // get interpolated gradient
          float x1 = (1.0-dx)*gradx[stc] + dx*gradx[stc+1];
          float x2 = (1.0-dx)*gradx[stcnx] + dx*gradx[stcnx+1];
          float x = (1.0-dy)*x1 + dy*x2; // interpolated x
          float y1 = (1.0-dx)*grady[stc] + dx*grady[stc+1];
          float y2 = (1.0-dx)*grady[stcnx] + dx*grady[stcnx+1];
          float y = (1.0-dy)*y1 + dy*y2; // interpolated y

          // show gradients
```

```
          ROS_DEBUG("[Path] %0.2f,%0.2f  %0.2f,%0.2f  %0.2f,%0.2f  %0.2f,%0.2f; final x=%.
                    gradx[stc], grady[stc], gradx[stc+1], grady[stc+1],
                    gradx[stcnx], grady[stcnx], gradx[stcnx+1], grady[stcnx+1],
                    x, y);

          // check for zero gradient, failed
          if (x == 0.0 && y == 0.0)
          {
            ROS_DEBUG("[PathCalc] Zero gradient");
            return 0;
          }

          // move in the right direction
          float ss = pathStep/hypot(x, y);
          dx += x*ss;
          dy += y*ss;

          // check for overflow
          if (dx > 1.0) { stc++; dx -= 1.0; }
          if (dx < -1.0) { stc--; dx += 1.0; }
          if (dy > 1.0) { stc+=nx; dy -= 1.0; }
          if (dy < -1.0) { stc-=nx; dy += 1.0; }

        }

        //      ROS_INFO("[Path] Pot: %0.1f  grad: %0.1f,%0.1f  pos: %0.1f,%0.1f\n",
        //           potarr[stc], x, y, pathx[npath-1], pathy[npath-1]);
      }

      //  return npath;                 // out of cycles, return failure
      ROS_DEBUG("[PathCalc] No path found, path too long");
      //savemap("navfn_pathlong");
      return 0;                 // out of cycles, return failure
    }


  //
  // gradient calculations
  //

  // calculate gradient at a cell
  // positive value are to the right and down
  float
    NavFn::gradCell(int n)
    {
      if (gradx[n]+grady[n] > 0.0)      // check this cell
        return 1.0;

      if (n < nx || n > ns-nx)  // would be out of bounds
```

```
        return 0.0;

      float cv = potarr[n];
      float dx = 0.0;
      float dy = 0.0;

      // check for in an obstacle
      if (cv >= POT_HIGH)
      {
        if (potarr[n-1] < POT_HIGH)
          dx = -COST_OBS;
        else if (potarr[n+1] < POT_HIGH)
          dx = COST_OBS;

        if (potarr[n-nx] < POT_HIGH)
          dy = -COST_OBS;
        else if (potarr[n+nx] < POT_HIGH)
          dy = COST_OBS;
      }

      else                              // not in an obstacle
      {
        // dx calc, average to sides
        if (potarr[n-1] < POT_HIGH)
          dx += potarr[n-1]- cv;
        if (potarr[n+1] < POT_HIGH)
          dx += cv - potarr[n+1];

        // dy calc, average to sides
        if (potarr[n-nx] < POT_HIGH)
          dy += potarr[n-nx]- cv;
        if (potarr[n+nx] < POT_HIGH)
          dy += cv - potarr[n+nx];
      }

      // normalize
      float norm = hypot(dx, dy);
      if (norm > 0)
      {
        norm = 1.0/norm;
        gradx[n] = norm*dx;
        grady[n] = norm*dy;
      }
      return norm;
    }


  //
  // display function setup
```

```
  // <n> is the number of cycles to wait before displaying,
  //     use 0 to turn it off

  void
    NavFn::display(void fn(NavFn *nav), int n)
    {
      displayFn = fn;
      displayInt = n;
    }


  //
  // debug writes
  // saves costmap and start/goal
  //

  void
    NavFn::savemap(const char *fname)
    {
      char fn[4096];

      ROS_DEBUG("[NavFn] Saving costmap and start/goal points");
      // write start and goal points
      sprintf(fn,"%s.txt",fname);
      FILE *fp = fopen(fn,"w");
      if (!fp)
      {
        ROS_WARN("Can't open file %s", fn);
        return;
      }
      fprintf(fp,"Goal: %d %d\nStart: %d %d\n",goal[0],goal[1],start[0],start[1]);
      fclose(fp);

      // write cost array
      if (!costarr) return;
      sprintf(fn,"%s.pgm",fname);
      fp = fopen(fn,"wb");
      if (!fp)
      {
        ROS_WARN("Can't open file %s", fn);
        return;
      }
      fprintf(fp,"P5\n%d\n%d\n%d\n", nx, ny, 0xff);
      fwrite(costarr,1,nx*ny,fp);
      fclose(fp);
    }
};
```