Algorithm 1 : Pseudo-code of the active contour used in the Method 2.

```
// Compute the average distance among the snake nodes
// used for the normalization of the continuity energy
averageDist = getAvgDistance();

// Compute the center of gravity used for the balloon energy
computeCG();

// Initial values for the energy
Energy_old = 0; Energy_new = 0;

// Create a Snake object from the approximated contour
// represented by a linked list of type Point {x, y}
MySnake = Snake( ApproximatedCountour );

// Loop for stages: at each stage the blur
// applied to the original image is reduced
do {
  // Counter variable for the number of adjusted snake nodes
  movedNodes = 0;

  // Count number of complete cycle through the snake nodes
  countIter = 0;

  // Initial values for the energy
  Energy_old = Energy_new;
  Energy_new = 0;

  // Create two auxiliar snake nodes for handle the
  // the last end node of the snake
  SnakeNode *nextNode = NULL, *prevNode = NULL;

  // Set position of the snakes to the initial position
  SnakeNode *start = MySnake.current;

  // For all snake nodes
  while (MySnake.current)
  {
    // Perform some special handling for prevNode and nextNode
    if (MySnake.current == start) {
      prevNode = (SnakeNode *) end;
      nextNode = (SnakeNode *) MySnake.current->next;

      if (++countIter > 1)
        break;

    } else {
      if (MySnake.current == end) {
        prevNode = (SnakeNode *) MySnake.current->prev;
        nextNode = (SnakeNode *) start;
      } else {
        prevNode = (SnakeNode *) MySnake.current->prev;
        nextNode = (SnakeNode *) MySnake.current->next;
      }
    }
```

```
// Get coordinates and energy of the coefficients for
// the current snake node
curX  = MySnake.current->data->GetXCoord();
curY  = MySnake.current->data->GetYCoord();
alpha = MySnake.current->data->GetAlpha ();
beta  = MySnake.current->data->GetBeta  ();
gamma = MySnake.current->data->GetGamma ();

// Compute the energy terms for all possible positions
// in the neighborhood
for (row = 0; row < NEIGHBOR_HEIGHT; row++)
  for (col = 0; col < NEIGHBOR_WIDTH; col++) {
    //
    // Compute indexes of the neighborhood
    //
    nIndex = row * NEIGHBOR_WIDTH + col;
    nX     = curX + (col - (NEIGHBOR_WIDTH  - 1) / 2);
    nY     = curY + (row - (NEIGHBOR_HEIGHT - 1) / 2);

    // Avoid neighborhood limits being out of range
    CheckAndCorrectLimits (nX, nY);

    // Get image gradient value in the position (nX, nY)
    I_grad[nIndex] = image->GetGradient (nX, nY);

    // Compute and store all energies in the position (nX, nY)
    // See equations 1.8, 1.9, and 1.10
    e_cont[nIndex] = Continuity (prevNode, nX, nY, nextNode);
    e_grad[nIndex] = Gradient   (prevNode, nX, nY, nextNode);
    e_ball[nIndex] = Balloon    (prevNode, nX, nY, nextNode);
  }

  // Normalize the energy to be in the range [0, 1]
  // See equations 1.11, 1.12, and 1.13
  normContinuity (e_cont, NEIGHBORHOOD);
  normGradient   (e_grad, NEIGHBORHOOD);
  normBalloon    (e_ball, I_grad, NEIGHBORHOOD);

  // Start with an insanely high upper bound
  minEnergy = MAX_ENERGY;

  // Now find the minimum energy location in the neighborhood
  for (int row = 0; row < NEIGHBOR_HEIGHT; row++)
    for (int col = 0; col < NEIGHBOR_WIDTH; col++) {
      //
      // Compute index and position in the neighborhood
      //
      nIndex = row * HOOD_WIDTH + col;
      nX     = curX + (col - (HOOD_WIDTH  - 1) / 2);
      nY     = curY + (row - (HOOD_HEIGHT - 1) / 2);

      // Check neighborhood limits
      CheckAndCorrectLimits (nX, nY);
```

```
          // Compute energy value of a node in the (nX, nY) position
          energy = alpha * e_cont[nIndex] + beta  * e_grad[nIndex];
                 + gamma * e_ball[nIndex];

          // Save position and energy value of the
          // point with minimum energy in the neighborhood
          if (energy < minEnergy) {
            minEnergy = energy;
            EminX = nX;
            EminY = nY;
          }
        }

      // Move current snake node to a position of minimum energy,
      // if one is found in the neighborhood
      if ( (curX != EminX) || (curY != EminY) ) {
        MySnake.current->data->SetPoint (EminX, EminY);
        movedNodes++;
        Energy_new += minEnergy;
      }

      // Move to the next node of the snake
      MySnake.moveToNext();
    }

    // Update average distance among the snake nodes
    // and center of gravity of the snake
    averageDist = getAvgDistance ();
    computeCG ();

    //  Selectively relaxes the curvature term for particular
    //  snake node if the node satisfies the following conditions:
    //  -node must have higher curvature than its neighboring nodes
    //  -node must have a curvature above the threshold value 0.25
    AllowCorners ();

    // Add or remove nodes based on the average and
    // curvature values of snake nodes
    Resampling ();

    // Reduce blur applied to the image if number of moved nodes
    // is less than 10% the number of the total nodes in the snake
    if ( (movedNodes < 0.1 * MySnake.getTotalNodes()) ||
         (Energy_old < Energy_new) && (stageCount  <= no_stages) )
    {
      stageCount++;
      if (stageCount <= no_stages)
      {
        image->ReduceBlur ();
        Energy_new = MAX_ENERGY;
      }
    }

} while (stageCount <= no_stages);
```