Finding the circle that best fits a set of points

L. MAISONOBE*

October 25th 2007

Contents

1 Introduction **2**

2 Solving the problem **2**

2.1 Principles 2

2.2 Initialization 3

2.3 Issues 4

2.4 Iterative improvement 4

2.4.1 Problem statement 4

2.4.2 Direct solving 5

2.4.3 Levenberg-Marquardt 5

2.4.4 Conjugate gradient 5

3 Numerical example **7**

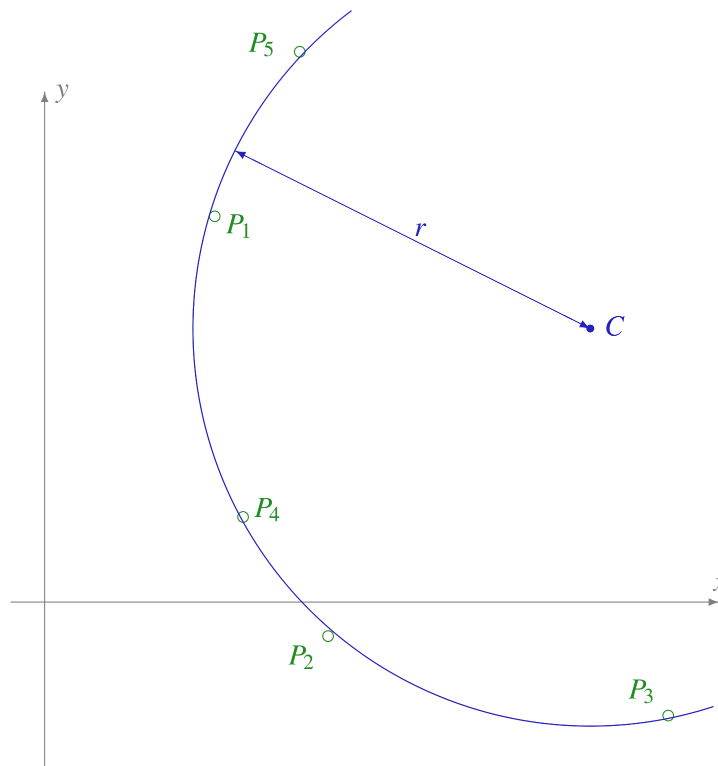
A Algorithms **10**

*luc@spaceroots.org

1 Introduction

We consider the following 2D problem: given a set of n points $P_i(x_i, y_i)$ $(i = 1 \ldots n)$ find the center $C(x_c, y_c)$ and the radius r of the circle that pass closest to all the points.

Figure 1: closest circle problem



For the latest version of this document, please check the downloads page of the spaceroots site at `http://www.spaceroots.org/downloads.html`. The paper can be browsed on-line or retrieved as a PDF, compressed PostScript or LaTeX source file. An implementation in the Java language of the algorithms presented here is also available in source form as a stand-alone file at `http://www.spaceroots.org/documents/circle/CircleFitter.java`.

2 Solving the problem

2.1 Principles

The underlying principles of the proposed fitting method are to first compute an initial guess by averaging all the circles that can be built using all triplets of non-

aligned points, and then to iteratively reduce the distance between the circle and the complete set of points using a minimization method.

2.2 Initialization

For any given triplet of non-aligned points, there is a single circle passing through all three points: the triangle circumcircle. We will use this property to build an initial circle.

Let $P_i(x_i, y_i)$, $P_j(x_j, y_j)$ and $P_k(x_k, y_k)$ be three points. The triangle circumcenter is at the intersection of the perpendicular bisectors of the triangle sides. Equation (1) holds as the circumcenter belongs to the perpendicular bisector of side (P_i, P_j) and equation (2) holds it also belongs to the perpendicular bisector of side (P_j, P_k):

$$\left\{\begin{aligned} x_{C_{i,j,k}} &= \frac{(x_i + x_j) + \alpha_{i,j}(y_j - y_i)}{2} \\ y_{C_{i,j,k}} &= \frac{(y_i + y_j) - \alpha_{i,j}(x_j - x_i)}{2} \end{aligned}\right. \tag{1}$$

$$\left\{\begin{aligned} x_{C_{i,j,k}} &= \frac{(x_j + x_k) + \alpha_{j,k}(y_k - y_j)}{2} \\ y_{C_{i,j,k}} &= \frac{(y_j + y_k) - \alpha_{j,k}(x_k - x_j)}{2} \end{aligned}\right. \tag{2}$$

Solving this set of linear equations is straightforward:

$$\left\{\begin{aligned} \alpha_{i,j} &= \frac{(x_k - x_i)(x_k - x_j) + (y_k - y_i)(y_k - y_j)}{\Delta} \\ \alpha_{j,k} &= \frac{(x_k - x_i)(x_j - x_i) + (y_k - y_i)(y_j - y_i)}{\Delta} \end{aligned}\right.$$
$$\text{where} \quad \Delta = (x_k - x_j)(y_j - y_i) - (x_j - x_i)(y_k - y_j) \tag{3}$$

Hence the circumcenter coordinates are:

$$\left\{\begin{aligned} x_{C_{i,j,k}} &= \frac{(y_k - y_j)(x_i^2 + y_i^2) + (y_i - y_k)(x_j^2 + y_j^2) + (y_j - y_i)(x_k^2 + y_k^2)}{2\Delta} \\ y_{C_{i,j,k}} &= -\frac{(x_k - x_j)(x_i^2 + y_i^2) + (x_i - x_k)(x_j^2 + y_j^2) + (x_j - x_i)(x_k^2 + y_k^2)}{2\Delta} \end{aligned}\right. \tag{4}$$

These coordinates can be computed as long as the determinant Δ is non-null, which means the three points are not aligned.

Since the method is intended to be general, we do not make any assumptions on the points. Some of them may have small (or even large) errors, some may be aligned with other ones, some may appear several times in the set. We cannot reliably select one triplet among the other ones. We build the initial guess of the fitting circle by averaging the coordinates of the circumcenters of all triplets of non aligned points.

This entire initialization process is implemented in algorithm 1 (page 10).

2.3 Issues

The initialization process described above may be inadequate in some cases.

If the number of points is very large, the number of triplets will be overwhelmingly large as it grows in n^3. In this case, only a subset of the complete triplets set must be used. Selecting the subset implies knowing the distribution of the points, in order to avoid using only points in the same area for initialization.

If the points sample contains really far outliers in addition to regular points, the initial center may be offset by a large amount. This is due to the fact the mean is not a robust statistic. This can be circumvented by either using a median rather than a mean or by dropping outliers (either before or after circumcenter determination).

If the points distribution is known to cover exactly one circle with a consistent step between the points and no outliers, a much simpler initialization process is simply to use the mean of the x and y coordinates for all points.

2.4 Iterative improvement

2.4.1 Problem statement

Once we have an initial guess for the circle center, we want to improve it for some definition of *best fit* against the points set. Using a least squares estimator based on the euclidean distance between the points and the circle is a common choice.

We try to minimize the cost function J:

$$J = \sum_{i=1}^{n} (d_i - r)^2 \quad \text{where} \quad d_i = \sqrt{(x_i - x)^2 + (y_i - y)^2}$$

d_i is the euclidean distance between the point $P_i(x_i, y_i)$ and the circle center $C(x, y)$ and r is the circle radius.

For a given circle center, we compute the optimum circle radius $\hat{r}$ by solving:

$$\left.\frac{\partial J}{\partial r}\right|_{r=\hat{r}} = 0 \Rightarrow \hat{r} = \frac{\sum_{i=1}^{n} d_i}{n} \tag{5}$$

This means that both d_i and $\hat{r}$ can be considered as functions of the circle center coordinates x and y which from now on will be the free parameters of our model.

Using the choice $r = \hat{r}$, the cost function can be rewritten:

$$J = \sum_{i=1}^{n} (d_i - \hat{r})^2 = \sum_{i=1}^{n} d_i^2 - \frac{\left(\sum_{i=1}^{n} d_i\right)^2}{n} \tag{6}$$

from this expression we can compute the gradient of the cost function with respect to the free parameters:

$$\nabla J \begin{cases} \dfrac{\partial J}{\partial x} = 2 \displaystyle\sum_{i=0}^{n} \frac{(x - x_i)(d_i - \hat{r})}{d_i} \\ \dfrac{\partial J}{\partial y} = 2 \displaystyle\sum_{i=0}^{n} \frac{(y - y_i)(d_i - \hat{r})}{d_i} \end{cases} \tag{7}$$

2.4.2 Direct solving

Equation (7) can easily be solved analytically. However experience shows that even for small errors with the initial guess, this leads to a solution $C(x,y)$ that is very far from the validity domain of the partial derivatives values.

This sensitivity to the initial guess is a common issue with GAUSS-NEWTON based least squares solvers. Iterating as in a fixed point method does not help either, iterations diverge once we are too far. So we have to refrain from using this easy but wrong solution and use a more robust approach.

2.4.3 Levenberg-Marquardt

The LEVENBERG-MARQUARDT method is a combination of the GAUSS-NEWTON and steepest descent methods. It benefits from the strength of both methods and is both robust even for starting points far from the solution and efficient near the solution. It is a sure bet and is highly recommended.

This method is however quite complex to implement, so it should be used only if a validated implementation of the method is easily available. The MINPACK implementation[1] in fortran is widely used. Our own Mantissa[2] Java library also provides a Java implementation based on the MINPACK one as of version 6.0. Many other implementations are available in various languages.

For cost function $J = \sum_{i=1}^{n}(d_i - \hat{r})^2$, the LEVENBERG-MARQUARDT method needs the jacobian matrix of the deviations $d_i - \hat{r}$. The two elements of row i of this matrix is given by equation (8):

$$\nabla(d_i - \hat{r}) = \begin{cases} \dfrac{\partial\left(d_i - \frac{1}{n}\sum_{k=0}^{n} d_k\right)}{\partial x} \\ \dfrac{\partial\left(d_i - \frac{1}{n}\sum_{k=0}^{n} d_k\right)}{\partial y} \end{cases}$$

$$= \begin{cases} \dfrac{x - x_i}{d_i} - \dfrac{1}{n}\displaystyle\sum_{k=0}^{n}\dfrac{x - x_k}{d_k} \\ \dfrac{y - y_i}{d_i} - \dfrac{1}{n}\displaystyle\sum_{k=0}^{n}\dfrac{y - y_k}{d_k} \end{cases} \tag{8}$$

This solver is far more efficient than the simple one we will present in the following section (by several orders of magnitude). The drawbacks are the implementation complexity.

2.4.4 Conjugate gradient

If no LEVENBERG-MARQUARDT based solver are available, we can still find the values of the free parameters that minimize the cost function J using the simpler

[1] http://www.netlib.org/minpack/lmder.f

[2] http://www.spaceroots.org/software/mantissa/index.html

conjugate gradient method with POLAK and RIBIÈRE factor. This will be less efficient than using LEVENBERG-MARQUARDT especially when the points are far from a real circle, but is very simple to do.

The conjugate gradient method is also an iterative one, using two embedded loops. The outer loop computes a new search direction by combining the cost function gradient at the current step and the previous search direction. The inner loop roughly minimizes the cost function in the search direction. This method is very easy to implement and allows to have a self-contained algorithm.

The rough minimization of the cost function along the search direction can be done by performing a few steps of a NEWTON solver on the derivative of the cost function $J(x+\lambda u, y+\lambda v)$ with respect to the step λ.

The following equations are the exact derivatives used for this rough inner minimization (we use $J_{u,v}(\lambda) = J(x+\lambda u, y+\lambda v)$, $d_{iu,v}(\lambda) = d_i(x+\lambda u, y+\lambda v)$ and $\hat{r}_{u,v}(\lambda) = \hat{r}(x+\lambda u, y+\lambda v)$ as a way to simplify the equation):

$$\left\{\begin{aligned}
J_{u,v}(\lambda) &= \sum_{i=0}^{n} d_{iu,v}^2(\lambda) - \frac{1}{n}\left(\sum_{i=0}^{n} d_{iu,v}(\lambda)\right)^2 \\
\frac{dJ_{u,v}(\lambda)}{d\lambda} &= 2\sum_{i=0}^{n} \frac{\left[(x+\lambda u - x_i)u + (y+\lambda v - y_i)v\right]\left[d_{iu,v}(\lambda) - \hat{r}_{u,v}(\lambda)\right]}{d_{iu,v}(\lambda)} \\
\frac{d^2 J_{u,v}(\lambda)}{d\lambda^2} &= 2(u^2+v^2)\sum_{i=0}^{n} \frac{d_{iu,v}(\lambda) - \hat{r}_{u,v}(\lambda)}{d_{iu,v}(\lambda)} \\
&\quad - \frac{2}{n}\left(\sum_{i=0}^{n} \frac{(x+\lambda u - x_i)u + (y+\lambda v - y_i)v}{d_{iu,v}(\lambda)}\right)^2 \\
&\quad + 2\hat{r}_{u,v}(\lambda)\sum_{i=0}^{n} \frac{\left((x+\lambda u - x_i)u + (y+\lambda v - y_i)v\right)^2}{d_{iu,v}^3(\lambda)}
\end{aligned}\right.$$

hence

$$\left.\frac{dJ_{u,v}(\lambda)}{d\lambda}\right|_{\lambda=0} = 2\sum_{i=0}^{n} \frac{\left[(x-x_i)u + (y-y_i)v\right]\left[d_i - \hat{r}\right]}{d_i} \tag{9}$$

and

$$\begin{aligned}
\left.\frac{d^2 J_{u,v}(\lambda)}{d\lambda^2}\right|_{\lambda=0} &= 2(u^2+v^2)\sum_{i=0}^{n} \frac{d_i - \hat{r}}{d_i} \\
&\quad - \frac{2}{n}\left(\sum_{i=0}^{n} \frac{(x-x_i)u + (y-y_i)v}{d_i}\right)^2 \\
&\quad + 2\hat{r}\sum_{i=0}^{n} \frac{\left((x-x_i)u + (y-y_i)v\right)^2}{d_i^3}
\end{aligned} \tag{10}$$

The preceding equations are used in a traditional conjugate gradient method, assuming x and y have been initialized, as shown in algorithm 2 (page 11).

The parameters of the algorithms are the maximal number of iterations $i_{\max}$, and the two convergence thresholds ε_{inner} and ε_{outer}. Their values depend on the problem at hand (number of points, expected error at each points, required accuracy). Finding the values for a given problem is a matter of trial and errors. As always with the conjugate gradient method, there is no real need to have an accurate minimum along the search direction, hence ε_{inner} may be quite large (I used values between 0.02 and 0.1). It is even possible to remove completely the convergence check and replace it with an arbitrary small number of NEWTON steps.

3 Numerical example

This section shows an example application of the algorithms described in this note. It is based on a simple case with only five points and an order of magnitude of about 1% of the circle radius for the errors in the points locations.

We start with the following set of five points:

point	x	y
P_1	30	68
P_2	50	-6
P_3	110	-20
P_4	35	15
P_5	45	97

Since there are five points in the set, there are ten different triplets, none of them have aligned points:

circumcenter	x	y	circumcenter	x	y
$C_{1,2,3}$	93.650	45.500	$C_{1,4,5}$	103.768	48.223
$C_{1,2,4}$	103.704	48.217	$C_{2,3,4}$	92.400	40.142
$C_{1,2,5}$	103.753	48.230	$C_{2,3,5}$	94.178	47.765
$C_{1,3,4}$	95.821	47.473	$C_{2,4,5}$	103.720	48.229
$C_{1,3,5}$	99.228	50.571	$C_{3,4,5}$	96.580	49.100

we observe that with this data set, four circumcenters ($C_{1,2,4}$, $C_{1,2,5}$, $C_{1,4,5}$ and $C_{2,4,5}$) are very close to each other. This is only a coincidence. This explains why few points show up in figure 2: the four rightmost points are almost superimposed.

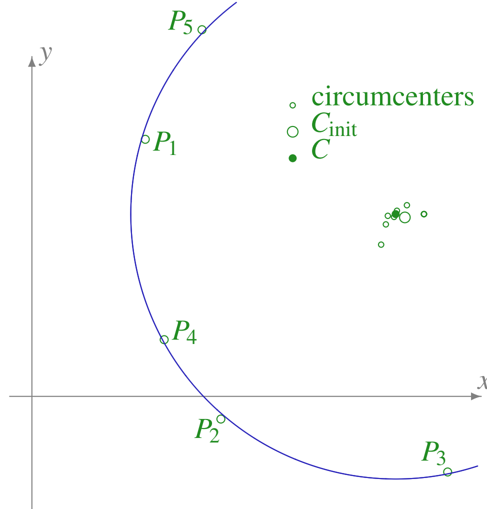
Averaging these circumcenters gives the initial value for the circle center:

$$C_{\text{init}}(98.680, 47.345)$$

The initial value of the cost function is 13.407.

Using the LEVENBERG-MARQUARDT method with very stringent convergence thresholds (10^{-10} for all tolerance settings: relative cost, relative parameters and

Figure 2: initialization



orthogonality), convergence is reached after only 4 cost function evaluations and 4 jacobians evaluations. Using the conjugate gradient method with a loose convergence setting for the inner loop and a tight one for the outer loop ($\varepsilon_{\text{inner}} = 0.1$ and $\varepsilon_{\text{outer}} = 10^{-12}$), convergence is reached after 7 iterations.

Both methods give the same results[3]. The minimal value found for the cost function is 3.127. The parameters for the best fitting circle and the corresponding residuals are:

$$\begin{cases} C(96.076, 48.135) \\ r = 69.960 \end{cases} \Rightarrow \begin{cases} d_1 - r = -0.963 \\ d_2 - r = 1.129 \\ d_3 - r = -0.417 \\ d_4 - r = -0.475 \\ d_5 - r = 0.725 \end{cases}$$

This example is only one example, no conclusions should be drawn from the similar results and iteration numbers. Some key features that depend on the problems and are not addressed in this paper are the following ones:

number of points: this example uses 5 points, other may have hundreds of points,

error level: errors range to about 1% of the circle radius, I have encountered real life applications were errors were around 0.001%, other were errors were about 30%,

[3] a previous version of this paper found slight differences between the results and a large difference in the number of iterations, this was due to an implementation bug in the conjugate gradient method

error distribution is homogeneous: but in some application errors may be highly correlated (dented or squizzed circles),

circle coverage: the example points all belong to the same left half of the circle, in some cases points will be regularly distributed over the circle, and in some cases there will be some strange distribution like 3.5 turns.

A Algorithms

```
σ_x ← 0
σ_y ← 0
q ← 0
loop for i = 1 to i = n − 2
   loop for j = i + 1 to j = n − 1
      loop for k = j + 1 to k = n
         compute Δ using equation (3)
         if |Δ| > ε then
            // we know the points are not aligned
            compute C_{i,j,k} using equation (4)
            σ_x ← σ_x + x_{C_{i,j,k}}
            σ_y ← σ_y + y_{C_{i,j,k}}
            q ← q + 1
         end if
      end loop
   end loop
end loop
if q = 0 then
   error all points are aligned
end if
return circle center C(σ_x/q, σ_y/q)
```

Algorithm 1: initialization

compute $\hat{r}$ using equation (5)
compute J using equation (6)
compute ∇J using equation (7)
if $|J| < \varepsilon_1$ **or** $|\nabla J| < \varepsilon_2$ **then**
  return *// we consider we already have the minimum*
end if
$J_{\text{prev}} \leftarrow J$
$\nabla J_{\text{prev}} \leftarrow \nabla J$
$\vec{u}_{\text{prev}} \leftarrow \vec{0}$
loop from $i = 1$ **to** $i = i_{\max}$

  // search direction
  $\vec{u} \leftarrow -\nabla J$
  if $i > 1$ **then**
    $\beta \leftarrow \nabla J^T (\nabla J - \nabla J_{\text{prev}}) / \nabla J_{\text{prev}}^2$ *// Polak-Ribiere coefficient*
    $\vec{u} \leftarrow \vec{u} + \beta \vec{u}_{\text{prev}}$
  end if
  $\nabla J_{\text{prev}} \leftarrow \nabla J$
  $\vec{u}_{\text{prev}} \leftarrow \vec{u}$

  // rough minimization along the search direction (a few Newton steps)
  loop
    $J_{\text{inner}} \leftarrow J$
    compute $dJ/d\lambda$ using equation (9)
    compute $d^2J/d\lambda^2$ using equation (10)
    $\lambda \leftarrow -(dJ/d\lambda)/(d^2J/d\lambda^2)$
    $C \leftarrow C + \lambda \vec{u}$
    compute $\hat{r}$ using equation (5)
    compute J using equation (6)
    compute ∇J using equation (7)
  while $i \leq i_{\max}$ **and** $|J - J_{\text{inner}}|/J \geq \varepsilon_{inner}$

  if $|J - J_{\text{prev}}|/J < \varepsilon_{outer}$
    return *// convergence reached*
  end if

end loop
error unable to converge in $i_{\max}$ iterations

Algorithm 2: conjugate gradient