CHAPTER 7

Iterative Solution of Systems of Equations

Numerical Methods in the Hydrological Sciences
Special Publication Series 57
Copyright 2005 by the American Geophysical Union
doi:10.1029/057SP12

7. Iterative Solution of Systems of Equations

7.1. Introduction

The methods for solving partial differential equations presented in Chapter 6 relied on Gaussian elimination (conveniently implemented with *MATLAB*'s "\" command) to solve the system of equations obtained when finite differences are used to approximate the derivatives. Gaussian elimination is not the only way to solve a system of equations. Iterative methods provide an alternative that can be advantageous. For example, when programming in one of the popular languages (FORTRAN, PASCAL, C), it is much easier to code the "loop" calculations needed for typical iterative solutions than it is to code Gaussian elimination. The matrix approach used in Chapter 6 is convienent for *MATLAB* but is not the way the calculations generally are done when working with a "non-matrix" language. [Consult the text by Wang and Anderson (1982) for more detail.] Also, some of the very large matrices that arise in complex hydrological problems are not solved directly very easily – iterative methods are actually preferred to Gaussian elimination. The basic idea of iterative solutions to equations was already presented in Chapter 2. In this chapter, we explore how iterative methods can be used to solve systems of equations.

7.2. Fixed-point iteration revisited

Recall that "fixed-point iteration" is one option for solving nonlinear equations [Chapter 2.6]. The equation is written in the form

$$x = g(x)$$

and the iteration formula is simply

$$x_{k+1} = g(x_k).$$

Convergence of the iteration is not guaranteed in general and the convergence criterion typically used is a check on whether x_{k+1} is "sufficiently close" to x_k.

Of course a linear equation is just a special case of a nonlinear equation. Although iteration doesn't make good sense computationally for a single linear equation, we can look at this case to motivate the actual use that interests us – solution of systems of linear equations. So, let's take the simple example

$$3x = 6.$$

To develop an iteration formula, we must "split" the 3. For example, we can say that $3x=2x+x$, subtract x from both sides of the equation, divide by 2, and have the iteration formula

$$x_{k+1} = \frac{-x_k}{2} + 3.$$

The first several steps in the iteration are

$$x_1 = \frac{-x_0}{2} + 3$$

$$x_2 = \frac{-x_1}{2}+3=\frac{-1}{2}\left(\frac{-x_0}{2}+3\right)+3=\left(\frac{-1}{2}\right)^2 x_0+3\left(1-\frac{1}{2}\right)$$

$$x_3 = \frac{-x_2}{2}+3=\left(\frac{-1}{2}\right)^3 x_0+3\left(1-\frac{1}{2}+\left(\frac{1}{2}\right)^2\right)$$

$$\vdots$$

$$x_{k+1}=\left(\frac{-1}{2}\right)^{k+1} x_0+3\left(1-\frac{1}{2}+\left(\frac{1}{2}\right)^2-\left(\frac{1}{2}\right)^3+\ldots+\left(\frac{1}{2}\right)^k\right).$$

We can see from inspection of the general equation above that the iteration converges to the right answer. The first term goes to zero as $k \to \infty$ (i.e., 1/2 to the k^{th} power approaches zero). The term in brackets is the geometric series. The limit of this series is [1/(1+1/2)]=2/3, so in the limit, x_k approaches 3*(2/3)=2.

There are other ways – in fact, an infinite number of ways – we can "split" 3 in the original equation. The general case can be represented as

$$x_{k+1}=\frac{-(3-a)}{a}x_k+\frac{6}{a}=\left[\frac{-(3-a)}{a}\right]^{k+1} x_0+\left(\frac{6}{a}\right)\left(1-\frac{3-a}{a}+\left(\frac{3-a}{a}\right)^2-\ldots+\left(\frac{3-a}{a}\right)^k\right).$$

We can see the requirement for convergence of the method. The term $-(3-a)/a$ must be less than one in magnitude. (Otherwise, as the term is raised to higher and higher powers, the iteration diverges.) This condition requires that a >3/2. Is there a "best" value for a? It should be clear from an examination of the steps above that the smaller our "iteration number", the faster the effect of the initial guess goes to zero and the faster the terms in the series goes to zero and hence the faster this series converges. Therefore, we want to have the absolute value of the "iteration number" $(3-a)/a$ be as small as possible. We can look at values for the iteration number as a varies from 3/2 to 10.

```
alpha=3/2:1/6:10;
iter=abs((3-alpha)./alpha);
plot(alpha,iter);
xlabel('value of the splitting parameter alpha')
ylabel('magnitude of the iteration number')
```

The graph that results from the *MATLAB* operations above (Fig. 7.1) shows that there is a "best" value. As we would have expected for this simple (simple-minded?) problem, 3 is the best splitting parameter because then no iteration at all is required! The example is nonetheless instructive because one can see that the progress of the iteration – how fast the process converges – depends on the choice for splitting. To make good use of information on whether a split is "good", we need to know something about the magnitude of the iteration number. We will return to this notion directly, but first, let's look at solutions for matrix-vector equations.

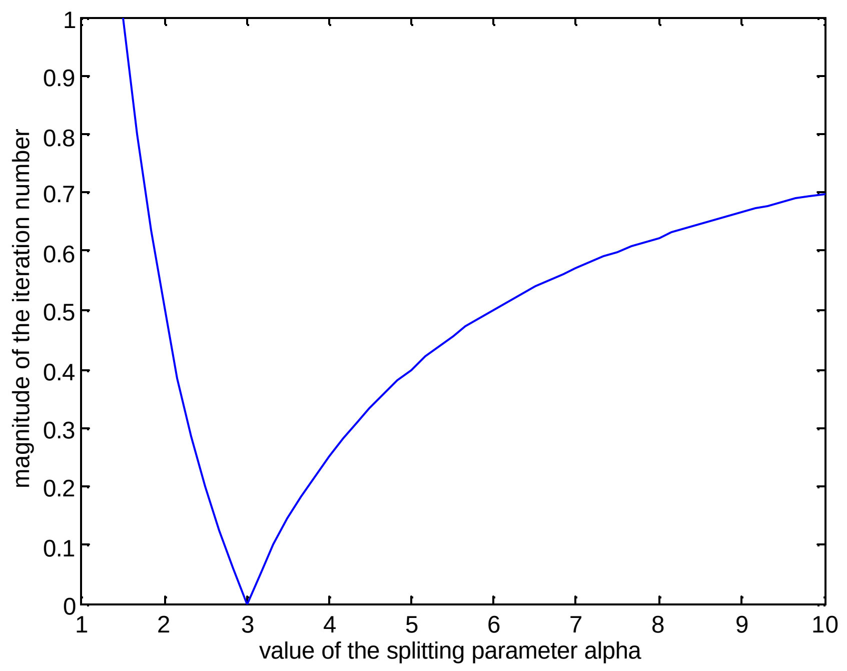

Figure 7.1. Splitting parameter for the example 3*x*=6.

7.3. Iterative solution of a system of equations

The notions above generalize to matrix-vector equations pretty well. Take a system of equations given by

$$Ax = b.$$

Split the matrix *A* into *D*+*B*. Rewrite the equation as follows,

$$Dx = -Bx + b$$

$$x = -D^{-1}Bx + D^{-1}b$$

which leads to the iteration formula

$$x_{k+1} = -D^{-1}Bx_k + D^{-1}b\,.$$

General iteration formulas have this form. The specifics differ because the choice of how to split the matrix differs. The simplest iteration (or at least one very simple one) uses the diagonal elements of matix *A* to form *D*. This method uses the sum of the part of *A* that is below the diagonal (*L*) and the part of *A* that is above the diagonal (*U*) as the *B* matrix. The iteration formula is therefore

$$x_{k+1} = -D^{-1}\left(L+U\right)x_k + D^{-1}b = \left[Q\right]^{k+1}x_0 + \left(I + Q + Q^2 + ... + Q^k\right)D^{-1}b,$$

where Q=D^{-1}(L+U). Note that because *D* is a diagonal matrix, D^{-1} is trivial to compute – it is just the inverse of the individual diagonal elements of *D*. This is a feature sought in iteration methods because computational ease is one of the reasons for using iterative methods. This simple iteration formula is known as *Jacobi* iteration. For finite-difference approximations to the Laplace equation [Chapter 6.3], this method "works", that is, it converges. Note that this means that $Q^k \rightarrow 0$ as $k \rightarrow \infty$. But how fast will it converge? And are there better methods? How can we judge these methods? For the single linear equation that we used as an example in the first section of these notes, it turned out that the magnitude of the "iteration number" was important. But now we have an iteration matrix, Q, rather than an iteration number. We need to examine the notion of the "magnitude" of a matrix before we proceed.

7.4. Vector and matrix norms

You are already familiar with the idea of magnitude applied to a vector. If the x component of velocity is 10 m s^{-1} and the y component is 10 m s^{-1}, you know that the magnitude of the velocity is about 14.14 m s^{-1}. You arrive at this answer by squaring the lengths of the x and y components, adding, and taking the square root. This measure of the magnitude of a vector is known as the L_2 norm. A "norm" is just the generalization of a measure of size.

The norm of a vector x is an associated, non-negative number, $\|x\|$, that satisfies the following requirements:

1) $\|x\| > 0$ for $x \neq 0$; $\|0\| = 0$
2) $\|cx\| = |c|\|x\|$
3) $\|x + y\| \leq \|x\| + \|y\|$

Here are the three most popular ways of assigning a vector norm.

i) $\|x\|_\infty = \max\limits_i |x_i|$ (the maximum component of the vector)
ii) $\|x\|_1 = |x_1| + |x_2| + \ldots + |x_n|$ (the sum of the lengths of all of the componenents)
iii) $\|x\|_2 = \sqrt{x_1^2 + x_2^2 + \ldots + x_n^2}$ (the "standard" norm with which you are familiar).

These are often referred to as the L_∞, L_1, and L_2 norms, respectively. These are the most commonly used forms of the more general L_p norm, $\|x\|_p = [x_1^p + x_2^p + x_3^p + \circ\circ\circ + x_n^p]^{1/p}$ [Box 7.1].

The norm of a square matrix A is a non-negative number, $\|A\|$, satisfying

1) $\|A\| > 0$ *for* $A \neq 0$; $\|Z\| = 0, \text{iff } Z = 0$
2) $\|cA\| = |c|\|A\|$
3) $\|A + B\| \leq \|A\| + \|B\|$
4) $\|AB\| \leq \|A\|\|B\|$

Because in many problems we have both matrices and vectors, it is convienenent to introduce the norm of a matrix in such a way that it will be consistent with a vector norm. A matrix norm is said to be compatible with a given vector norm if

$\|Ax\| \le \|A\|\|x\|$.

A matrix norm constructed to be compatible is said to be subordinate to the given vector norm.

For the three "popular" vector norms given above, the subordinate matrix norms are:

i) $\|A\|_\infty = \max_i \sum_{k=1}^{n} |a_{ik}|$, (the maximum sum of row elements)

ii) $\|A\|_1 = \max_k \sum_{i=1}^{n} |a_{ik}|$, (the maximum sum of column elements)

iii) $\|A\|_2 = \sqrt{l_1}$, where l_1 is the largest eigenvalue of the matrix $A'A$.

Next consider how matrix-vector norms are useful in examining iteration. Here are three useful theorems about matrix norms (given without proof).

Theorem 1. In order that $A^m \to 0$ as $m \to \infty$, a necessary and sufficient condition is that the eigenvalues of A be less than unity in magnitude.

Theorem 2. No eigenvalue of a matrix exceeds any of its norms in modulus. (This means that, from Theorem 1, sequential powers of a matrix will converge to zero if any matrix norm is <1.)

Theorem 3. In order that the series $I + A + A^2 + ... + A^m + ...$ converge, it is necessary and sufficient that $A^m \to 0$. In this case, the sum of the series converges to $(I\text{-}A)^{-1}$.

Armed with this knowledge, we go back to Jacobi iteration,

$$x_{k+1} = -D^{-1}(L+U)x_k + D^{-1}b = [Q]^{k+1}x_0 + \left(I+Q+Q^2+...+Q^k\right)D^{-1}b,$$

from which we see that the method converges if $\|Q\| < 1$. Note that, in direct analogy with the trivial example that we used for motivation – iterative solution of a single linear equation – the rate of convergence of the matrix iteration problem is determined by the "size" of the iteration matrix. The smaller the norm of Q, the faster will the method converge. The aim of improved methods for solving linear equations iteratively is to reduce the norm of the iteration matrix. In particular, because it is the maximum eigenvalue of the iteration matrix that controls convergence, "good" iterative methods will have iteration matrices with small eigenvalues.

7.5. Iterative methods for finite difference equations

Recall the example problem from Chapter 6.3 (for the Laplace equation).

```
M_diag=sparse(1:21,1:21,-4,21,21);
L_diag1=sparse(2:21,1:20,1,21,21);
L_diag2=sparse(8:21,1:14,1,21,21);
L_diag1(8,7)=0; L_diag1(15,14)=0;
A=M_diag+L_diag1+L_diag2+L_diag1'+L_diag2';
```

Remember that these steps give the blocked matrix with -4's on the main diagonal and 1's on a few off-diagonals.

```
full(A(1:9,1:9))

ans =
    -4     1     0     0     0     0     0     1     0
     1    -4     1     0     0     0     0     0     1
     0     1    -4     1     0     0     0     0     0
     0     0     1    -4     1     0     0     0     0
     0     0     0     1    -4     1     0     0     0
     0     0     0     0     1    -4     1     0     0
     0     0     0     0     0     1    -4     0     0
     1     0     0     0     0     0     0    -4     1
     0     1     0     0     0     0     0     1    -4
```

The right-hand vector for this example problem has three values of -100.

```
b=zeros(21,1); b(7)=-100; b(14)=-100; b(21)=-100;
```

Now rather than use the *MATLAB* backslash command to get the solution, suppose we use the Jacobi iteration approach. The matrix D is just what we defined as `M_diag` in the *MATLAB* statements above. The $L+U$ portion is everything else; we can define them in *MATLAB* as

```
L=L_diag1+L_diag2; U=L'; LnU=L+U;
```

To get D^{-1}, just write `Dinv=inv(M_diag)`. The iteration matrix is `Q=-Dinv*LnU`. We solve the equations using a loop that ends when the fractional change in h is less than 10^{-3}.

```
convcrit=1e9;
h_old=ones(21,1);
kount=0;
while convcrit>1e-3
   kount=kount+1;
   h=Q*h_old+Dinv*b;
   convcrit=max(abs(h-h_old)./h);
   h_old=h;
end
```

It takes 40 iterations to reduce the relative change in the iterative values to $< 10^{-3}$. Compare the results from this iteration (below) with the results in Chapter 6.3 obtained with the "\" command.

```
output=[h(1:7) h(8:14) h(15:21)]

output =
    0.3519    0.4973    0.3519
    0.9112    1.2865    0.9112
    2.0076    2.8287    2.0076
    4.2929    6.0153    4.2929
    9.1505   12.6501    9.1505
   19.6612   26.2865   19.6612
   43.2090   53.1759   43.2090
```

What about the rate of convergence? A good iteration method will have a norm that is "small". The closer to unity the norm, the slower will be the rate of convergence. For the Jacobi iteration matrix for the example problem, `abs(max(eig(Q)))` gives 0.815. For the small example problem this isn't horrendous, but as the size of the problem gets bigger (the mesh spacing for a fixed area gets finer), the norm of the Jacobi iteration matrix gets close to one very quickly. The method always converges, but progress can be agonizingly slow.

The method known as "successive over-relaxation" (SOR) is the simplest of the useful iterative methods for solving systems of linear equations. The iteration formula is

$$x_{k+1} = S(w)\,x_k + \left(w^{-1}D + L\right)^{-1} b,$$

where the iteration matrix, S, is

$$S(w) = -(w^{-1}D + L)^{-1}\left(U + (1 - w^{-1})D\right).$$

The Greek letter w in the iteration formula is an iteration parameter, chosen to accelerate convergence. The identification of an optimal value for w is not easy for complex problems. Theoretically, the optimal value of w for the Laplace equation is

$$w_{opt} = \frac{2}{1 + \sqrt{1 - r^2(Q)}},$$

where $r(Q)$ is the magnitude of the largest eigenvalue of the Jacobi iteration matrix.

For the example problem, we can calculate w_{opt} using 0.815 for $r(Q)$, a value that can be found using the *MATLAB* `normest` command, `wopt=2/(1+sqrt(1-(normest(Q))^2))`, which gives a value of about 1.27. [Note that the approximate norm will fail to give a usable estimate when the maximum eigenvalue of the Jacobi matrix is near unity. In this case, the command in the example code, `max(eig(full(S))`, should be used.] We can apply the SOR method to the example problem:

```
y=inv(D*(1/wopt)+L);
S=-y*(U+(1-(1/wopt))*D);
convcrit=1e9;
h_old=ones(21,1);
kount=0;
while convcrit>1e-3
   kount=kount+1;
   h=S*h_old+y*b;
   convcrit=max(abs(h-h_old)./h);
   h_old=h;
end
```

For the SOR method with w_{opt} =0.815, the convergence criterion is satisfied after 14 iterations. The results are,

```
output=[h(1:7) h(8:14) h(15:21)]

output =
    0.3529    0.4988    0.3530
    0.9131    1.2893    0.9132
    2.0103    2.8323    2.0103
    4.2957    6.0194    4.2957
    9.1532   12.6538    9.1532
   19.6632   26.2894   19.6632
   43.2101   53.1774   43.2101
```

How "big" is the iteration matrix for the SOR method? `abs(max(eig(full(s))))`

```
for i=1:max(size(w))
   y=inv(D*(1/w(i))+L);
   S=-y*(U+(1-(1/w(i)))*D);
   rho(i)=abs(max(eig(full(S))));
end
plot(w,rho)
xlabel('SOR iteration parameter')
ylabel('Magnitude of maximum eigenvalue of the iteration matrix')
```

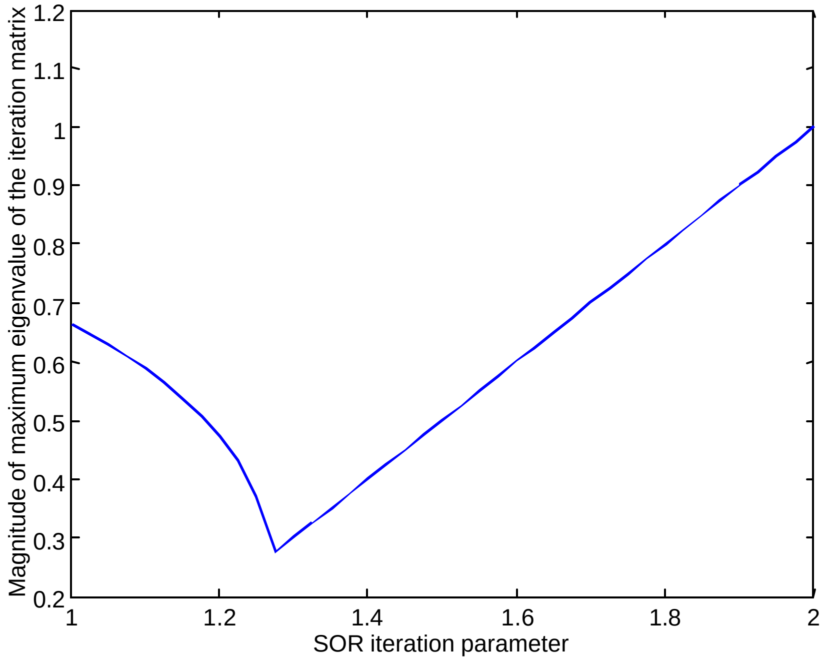

Figure 7.2. Splitting parameter for the SOR example.

Compare this figure with the one we drew for our simple-minded, one-linear-equation example. The analogy isn't perfect by any means, but the outcome is the same – there is an optimum value of an iteration parameter that speeds convergence of the iterative procedure.

The big trick is, of course, to choose a good iteration parameter. Sometimes it is sensible to calculate the maximum eigenvalue of the Jacobi matrix, but that procedure is in no case the "universal" solution. If you need to pursue this topic further, you will have to consult a text on numerical solution of systems of linear equations. Although the text is a bit "long in the tooth" now, you might try Remson et al. (1971) for a start.

The iterative methods that are used most widely today are more efficient (smaller matrix norms) than the very simple methods presented above (including SOR). One of these methods is called the Strongly Implicit Procedure (SIP). MODFLOW [Box 6.2] uses SIP to solve the groundwater flow equations (McDonald and Harbaugh, 1988). The SIP method uses a handful of iteration parameters; choosing them is an art, although the performance of the method does not depend critically on having exact values.

Another relatively new method is the conjugate-gradient method (e.g., Hill, 1990). The conjugate-gradient technique is not only powerful, but the acceleration parameter is calculated as part of the solution so no special tricks are needed to make it work efficiently. The conjugate-gradient method is really an iterative method for finding the minimum of a nonlinear function. Its use in solving sparse linear systems stems from recognizing that the minimum of

$$f(x) = \frac{1}{2}\left|Ax - b\right|^2$$

is unique and occurs for a value of x that satisfies the equation $Ax=b$. *MATLAB* has several built-in conjugate-gradient solvers, including `cgs`, which uses the "conjugate gradients squared" method. The syntax is `x=cgs(A,b)` where $Ax=b$ is the system to be solved (A is an $n\times n$ square matrix and b has length n). Error tolerance and maximum number of iterations can be set as described in "`help cgs`". Another *MATLAB* conjugate-gradient solver is `pcg`, which uses a preconditioned conjugate gradient method. However, for `pcg` the $n\times n$ coefficient matrix A must be positive definite and symmetric, which is often not true of the coefficient matrices in finite difference solutions to partial differential equations.

7.6. Problem

For the problem on the Brookhaven landfill from Chapter 6.7, use the Jacobi, SOR, and conjugate-gradient methods to solve the finite-difference equations. Write your own code for the Jacobi and SOR methods; use *MATLAB*'s `cgs` function for the conjugate-gradient method. Compare the number of iterations needed to achieve a solution using the different methods. (The number of iterations can be set in `cgs`.) Examine the effect of changing the value of the convergence criterion. Examine the effect of changing grid spacing. (Note that *MATLAB's* preconditioned conjugate gradients solver, `pcg`, will not work for this case because the coefficient matrix is not symmetric.)

7.7. References

Hill, M.C., Preconditioned conjugate-gradient 2(PCG2), a computer program for solving ground-water flow equations. U.S. Geological Survey Water-Resources Investigations Report 90-4048, 1990.

McDonald, M.G. and A.W. Harbaugh, A modular three-dimensional ground-water flow model. U.S. Geological Survey Techniques of Water-Resources Investigations, Book 6, Chap. A1, 586 pp., 1988.

Remson, I., Hornberger, G.M., and F.J. Molz, *Numerical Methods in Subsurface Hydrology*, 389 pp., Wiley-Interscience, New York, 1971.

Wang, H.F. and M.P. Anderson, *Introduction to Groundwater Modeling: Finite Difference and Finite Element Methods*, 237 pp., W.H. Freeman, San Francisco, 1982.

Box 7.1. L_p norms and circles

The general equation for the L_p norm is

$$\|x\|_p = [x_1^p + x_2^p + x_3^p + \circ\circ\circ + x_n^p]^{1/p}$$

As noted in the text, p=2 gives the “standard” distance; p=1 gives the sum of absolute values; and $p=\infty$ gives the maximum component. Other values of p don’t have as intuitive a meaning. One interesting way to think of these norms is in terms of the associated circles. In two dimensions, the line of constant value of a norm is a circle. The L_2 (Euclidean) norm produces the circle we are used to. But other norms also correspond to "circles." We can view these "circles" using the following m-file.

```
function y=isoLp(p)
x=[];y=[];
x=0:0.01:1;
y=(1-x.^p).^(1/p);
plot(x,y,-x,y,'b',x,-y,'b',-x,-y,'b')
axis('square')
```

The figure below shows the resulting "circles" for four values of p.

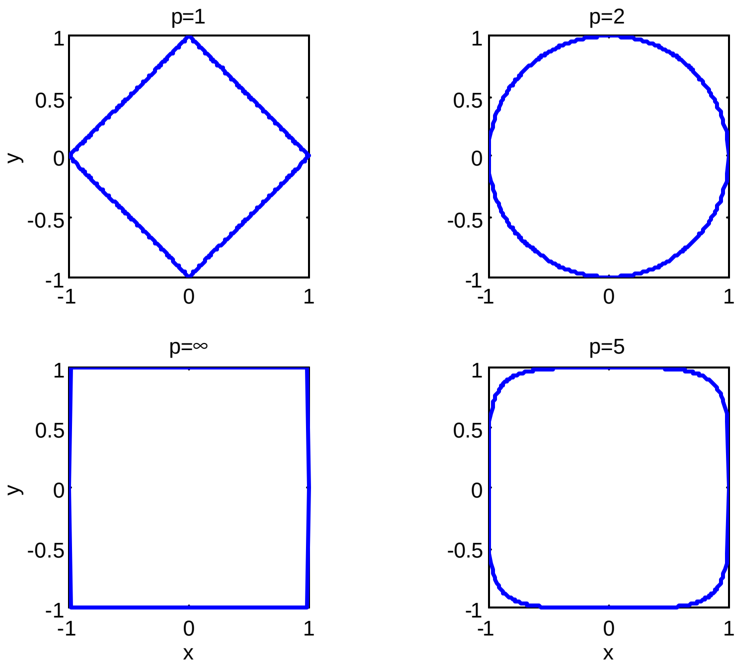

Figure B7.1. “Circles” associated with the L_p norm for several values of p.

Box 6.2. MODFLOW

The most widely used computer program in the world for simulating groundwater flow MODFLOW, a modular code developed by the U.S. Geological Survey in the early 1980's and continually improved since then (McDonald and Harbaugh, 1988; Harbaugh et al., 2000). The code along with documentation can be downloaded from the USGS Web site.

MODFLOW uses the finite difference method to solve the groundwater equations using the block centered node approach. An aquifer system is divided into rectangular blocks by a grid organized by rows, columns, and layers. Each block is called a "cell." Hydraulic properties are assigned to the cells, and boundary conditions are set by specifying cells to be constant head, no-flow, and so forth (see Figure B6.2.1).

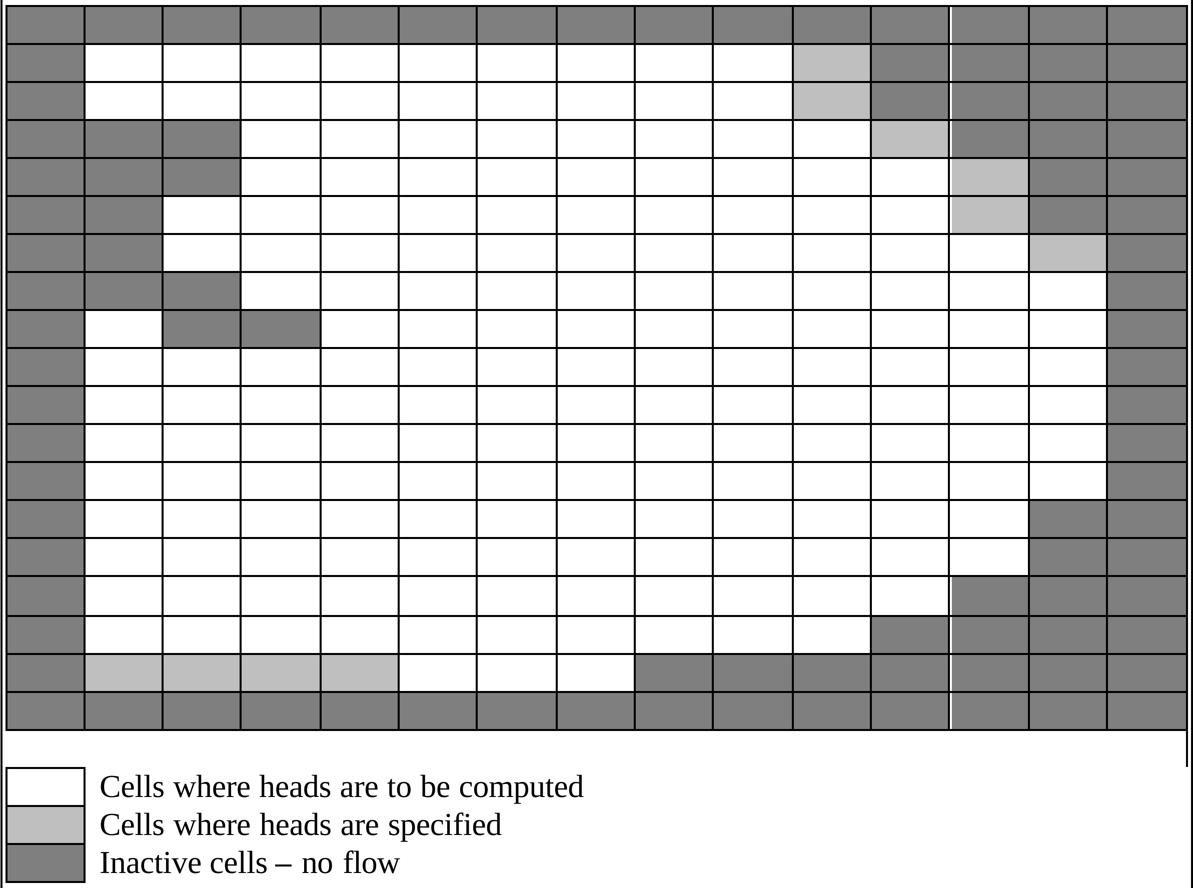
□ Cells where heads are to be computed
▒ Cells where heads are specified
■ Inactive cells – no flow

Figure B6.2.1. Example of a MODFLOW grid. Finite difference equations are written for cell-centered nodes.