13
Numerical Linear Algebra

We consider here the numerical side of linear algebra, the symbolic side being described in Chapter 8. The linear algebra numerical analysis and methods are discussed in [TBI97, Sch02]. The book of Golub and Van Loan [GVL12] is a major reference in this domain.

Numerical linear algebra plays a key role in *scientific computing* including the numerical solution of ordinary and partial differential equations, optimisation problems, and signal processing problems.

The numerical resolution of several of these problems, even linear ones, relies on algorithms made from nested loops; at the bottom of these loops, in most cases a linear system is solved. Non-linear algebraic systems are often solved using Newton's method: here again, we have to deal with linear systems. Efficiency and robustness of numerical linear algebra methods are thus crucial.

This chapter is split into three sections: in the first, we discuss the different sources of inexactness of numerical linear algebra computations; the second section (§13.2) discusses, without trying to be exhaustive, the more classical problems (linear system resolution, eigenvalue computation, least squares); in the third section (§13.3) we consider the case of sparse matrices. This last section is not only a guide for the user, but also an introduction to methods which are part of an active research domain.

13.1 Inexact Computations

We consider here classical linear algebra problems (linear system resolution, eigenvalue and eigenvector computation, etc.) which are solved using floating-point (i.e., inexact) computations. The different kinds of floating-point numbers available in Sage are detailed in Chapter 12.

Consider for example the system $Ax = b$ to solve, where A is a matrix with real coefficients. How big error δx do we get if we modify A by δA or b by δb? We give some partial answers in this chapter.

13.1.1 Matrix Norms and Condition Number

Let $A \in \mathbb{R}^{n\times n}$ (or $\mathbb{C}^{n\times n}$). We define on $\mathbb{R}^n$ (or $\mathbb{C}^n$) a norm $\|x\|$, for example the norm $\|x\|_\infty = \max |x_i|$ or $\|x\|_1 = \sum_{i=1}^n |x_i|$, or even the Euclidean norm $\|x\|_2 = (\sum_{i=1}^n |x_i|^2)^{1/2}$; then the quantity

$$\|A\| = \max_{\|x\|=1} \|A\ x\|$$

defines a norm on the set of $n \times n$ matrices. We say it is an *induced norm* with respect to the norm defined on $\mathbb{R}^n$ (or $\mathbb{C}^n$). The *condition number* of a nonsingular matrix A is defined by $\kappa(A) = \|A^{-1}\| \cdot \|A\|$. The fundamental result is that, if A is slightly modified by δA and b by δb, then the solution x of the linear system $Ax = b$ is modified by δx satisfying

$$\frac{\|\delta x\|}{\|x\|} \le \frac{\kappa(A)}{1 - \kappa(A)\|\delta A\|/\|A\|} \left(\frac{\|\delta A\|}{\|A\|} + \frac{\|\delta b\|}{\|b\|} \right).$$

The norms $\|\cdot\|_\infty$ and $\|\cdot\|_1$ are easy to compute: $\|A\|_\infty = \max_{1\le i\le n}(\sum_{j=1}^n |a_{ij}|)$ and $\|A\|_1 = \max_{1\le j\le n}(\sum_{i=1}^n |a_{ij}|)$. On the contrary, the norm $\|\cdot\|_2$ is more difficult to compute, since $\|A\|_2 = \sqrt{\rho(A^tA)}$, the spectral radius ρ of a matrix A being the largest modulus of its eigenvalues.

The Frobenius norm is defined by

$$\|A\|_F = \left(\sum_{i=1}^n \sum_{j=1}^n |a_{ij}|^2 \right)^{1/2}.$$

We easily check that $\|A\|_F^2 = \mathrm{trace}(A^tA)$. Contrary to the above norms, it is not an induced norm.

Computing Matrix Norms in Sage. The matrices in Sage have a `norm(p)` method. According to the value of the argument p it computes:

$p = 1$:	$\|A\|_1$,	$p = 2$:	$\|A\|_2$,
$p =$ `Infinity` :	$\|A\|_\infty$,	$p =$ `'frob'` :	$\|A\|_F$.

This method only works when the matrix coefficients can be converted into complex floating-point numbers `CDF`. Please note that we write `A.norm(Infinity)` but `A.norm('frob')`. With `A.norm()` we obtain the default norm $\|A\|_2$.

Errors and Condition Number: an Example, with Exact and Approximate Computations. Let us use the capability of Sage to perform exact computations when the coefficients are rational. We consider the Hilbert matrix

$$a_{ij} = 1/(i+j-1),\ i,j = 1,\ldots,n.$$

The following program computes[1] exactly the condition number of Hilbert matrices, using the norm $\|\cdot\|_\infty$:

```
sage: def cond_hilbert(n):
....:     A = matrix(QQ, [[1/(i+j-1) for j in [1..n]] for i in [1..n]])
....:     return A.norm(Infinity) * (A^-1).norm(Infinity)
```

Here are the results for a few n:

n	condition number
2	27
4	28375
8	33872791095
16	5.06277e+22
32	1.35711e+47

We see the extremely fast growth of the condition number with respect to n. We can prove that $\kappa(A) \simeq e^{7n/2}$, which indeed grows very quickly. Still computing in the rational field, we take $x = [1,\cdots,1]^t$, $y = Ax$ and we solve the system $\widetilde{A}s = y$ where $\widetilde{A}$ is a slight modification of the matrix A; then we compare the solution s and $x = A^{-1}y$, thus measuring the error introduced by the modification of A:

```
sage: def diff_hilbert(n):
....:     x = vector(QQ,[1 for i in range(0,n)])
....:     A = matrix(QQ, [[1/(i+j-1) for j in [1..n]] for i in [1..n]])
....:     y = A*x
....:     A[n-1,n-1] = (1/(2*n-1))*(1+1/(10^5))   # modifies the matrix
....:     s = A\y
....:     return max(abs(float(s[i]-x[i])) for i in range(0,n))
```

We obtain:

n	error (`diff_hilbert`)
2	3.99984e-05
4	5.97610e-3
8	4.80536
16	67.9885
32	20034.3

The error very rapidly becomes too large with respect to $x_i = 1$.

Let us now perform the same computation with floating-point coefficients. We no longer modify the matrix A, but the floating-point arithmetic will perform

[1] In Sage, there exists a `condition()` method which returns the condition number for various norms, but it exists for matrices with coefficients in `RDF` or `CDF`.

small rounding errors (thus, the matrix A is actually *not* exactly the Hilbert matrix any more). We perform again the above computation: the vector y being first computed by rounding the exact values in `RDF`, we try to recover x by solving the linear system $Ax = y$:

```
sage: def hilbert_diff(n):
....:     j = var("j")
....:     f = lambda i: sum(1/(i+j-1),j,1,n)
....:     y = vector(RDF, [f(i+1) for i in range(0,n)])
....:     A = matrix(RDF, [[1/(i+j-1) for i in [1..n]] for j in [1..n]])
....:     x = A.solve_right(y)
....:     return max(abs(x[i]-1.0) for i in range(0,n))
```

We also compute the condition number $\kappa(A)$. According to n we obtain:

n	error (`hilbert_diff`)	$\kappa(A)$	$\kappa(A)$ /error
2	6.66134e-16	27.0	2.46716e-17
4	2.38586e-13	28375.0	8.40835e-18
8	3.45957e-07	3.38723e+10	1.02134e-17
16	73.9841	1.00834e+19	7.33725e-18
32	76.6078	1.22870e+19	1.91195e-18

We see for example that for $n = 16$, the error made on the solution (with the infinite norm) is so large that all digits of the result are wrong (with the particular choice of x we have made, $x_i = 1$, the absolute and relative errors coincide).

Remarks. But why then compute with floating-point numbers? The performance is not always the only reason since efficient linear algebra libraries with rational arithmetic exist (for example Linbox, used by Sage); these libraries implement algorithms which are slower than their floating-point equivalent, but could be used for the resolution of moderate size linear systems. However, a second source of inexactness comes from the fact that, in real applications, the coefficients are usually not known (or measured) exactly. For example, solving a non-linear system of equations using Newton's method will naturally involve inexact terms.

Ill-conditioned linear systems (if we except extreme cases like Hilbert matrix) are more the rule than the exception: we often encounter (in physics, chemistry, biology, etc.) systems of ordinary differential equations of the form $du/dt = F(u)$ where the Jacobian matrix $DF(u)$, defined by partial derivatives $\partial F_i(u)/\partial u_j$, defines an ill-conditioned system. The eigenvalues span a very large range, which yields a bad condition number of $DF(u)$; this corresponds to the fact that the system models phenomena of different time scales. Unfortunately, in practice, we have to solve linear systems whose matrix is $DF(u)$.

All the computations (matrix decomposition, computation of eigenelements, convergence of iterative methods) depend on an appropriately defined condition number. We therefore should keep this notion in mind as soon as we perform linear algebra computations with floating-point numbers.

13.2 Dense Matrices

13.2.1 Solving Linear Systems

Methods to Avoid. Using Cramer's formula should (almost) always be avoided. A recurrence reasoning shows that computing the determinant of an $n \times n$ matrix using Cramer's formula requires on the order of $n!$ multiplications and additions. To solve a system of size n, we have to compute $n + 1$ determinants. Consider $n = 20$:

```
sage: n = 20; cost = (n+1)*factorial(n); cost
51090942171709440000
```

we obtain the huge value of 51 090 942 171 709 440 000 multiplies. Assuming our computer performs $3 \cdot 10^9$ multiplications per second, let us find how long the computation would last:

```
sage: v = 3*10^9
sage: print("%3.3f" % float(cost/v/3600/24/365))
540.028
```

The computation would thus require about 540 years! Of course, we can use Cramer's formula to solve a 2×2 system, but not much beyond! All methods used in practice have in common a polynomial cost in the dimension, i.e., of order n^p, with p small ($p = 3$ in general).

Practical Methods. The solution of linear systems $Ax = b$ is in most cases based on a factorisation of the matrix A into a product of two matrices $A = M_1 M_2$, where M_1 and M_2 correspond to "easy" linear systems. To solve $Ax = b$, we thus first solve $M_1 y = b$, then $M_2 x = y$.

For example M_1 and M_2 can be triangular matrices; in this case, once the factorisation is performed, we have to solve two triangular linear systems. The factorisation is much more expensive than solving the triangular linear systems (for example $O(n^3)$ for the LU factorisation against $O(n^2)$ for the triangular linear systems). When several systems with the same matrix have to be solved, we should therefore perform the matrix decomposition only once. Of course, we *never* invert a matrix to solve a linear system, since the inversion requires the matrix factorisation, followed by solving n systems instead of only one.

13.2.2 Direct Resolution

The simplest way to solve a linear system is illustrated by

```
sage: A = matrix(RDF, [[-1,2],[3,4]])
sage: b = vector(RDF, [2,3])
sage: x = A\b; x
(-0.20000000000000018, 0.9000000000000001)
```

Within Sage, matrices have a method `solve_right` to solve linear systems (which is based on the LU decomposition); this method is used above. We could also write:

```
sage: x = A.solve_right(b)
```

The `x = A\b` syntax is similar to what can be found in the Matlab, Octave and Scilab systems.

13.2.3 The LU Decomposition

```
sage: A = matrix(RDF, [[-1,2],[3,4]])
sage: P, L, U = A.LU()
```

This method gives the L and U factors, and the permutation matrix P, such that $A = PLU$ (or equivalently, $P^tA = LU$). The matrix L is lower triangular, with unit diagonal, and U is upper triangular. The matrix P depends on the pivot choices. For this, the strategy described in §8.2.1 consists, at step k, in finding an invertible coefficient a_{ik} in column k, and to use it as pivot. But here, this strategy must be improved, as we compute with floating-point numbers. To demonstrate this, let us consider the following system:

$$\begin{cases} \varepsilon x + y &= 1, \\ x + y &= 2. \end{cases}$$

If we use the coefficient ε of x in the first equation as pivot, we obtain $y = (1-2\varepsilon)/(1-\varepsilon)$ and $x = 1/(1-\varepsilon)$. Let us choose a small number for ε:

```
sage: eps = 1e-16
sage: y = (1-2*eps)/(1-eps)
sage: x = (1-y)/eps
sage: x, y
(1.11022302462516, 1.00000000000000)
```

The solution is very inaccurate! This is a consequence of the fact that, in the set of floating-point numbers we use:

```
sage: 1. + eps == 1.
True
```

as `RR(1).ulp()`$> \varepsilon$ (see §11.3.1). Now, we choose as pivot the coefficient of x in the second equation (that is to say we permute the equations), we get:

$$\begin{cases} x + y &= 2, \\ (1-\varepsilon)y &= 1 - 2\varepsilon. \end{cases}$$

We obtain:

```
sage: y = (1-2*eps)/(1-eps)
sage: x = 2-y
sage: x, y
(1.00000000000000, 1.00000000000000)
```

which is much more acceptable.

To obtain the PLU factorisation of A, the partial pivoting by column algorithm is used: at the first step we choose in the first column the coefficient a_{i1} of maximum absolute value and exchange row i and row 1; then, we fill the first column (except the first element) with zeros using the Gaussian elimination process. The application to the next steps is obvious.

Please note that Sage keeps in memory the factorisation of A: the `A.LU_valid()` command answers `True` if and only if the LU factorisation has already been computed. Moreover, the `A.solve_right(b)` command will only compute the factorisation if required, i.e., if it has not been computed before, or if the matrix A has changed.

EXAMPLE. Let us create a random matrix of size 1000, and two size-1000 vectors:

```
sage: A = random_matrix(RDF, 1000)
sage: b = vector(RDF, range(1000))
sage: c = vector(RDF, 2*range(500))
```

Let us first solve the system $Ax = b$:

```
sage: %time x = A.solve_right(b)
CPU times: user 132 ms, sys: 4 ms, total: 136 ms
Wall time: 140 ms
```

and now let us solve $Ay = c$:

```
sage: %time y = A.solve_right(c)
CPU times: user 68 ms, sys: 0 ns, total: 68 ms
Wall time: 72.8 ms
```

The second resolution is faster, because it used the LU factorisation computed in the first one.

13.2.4 The Cholesky Decomposition

A symmetric matrix A is said to be *positive definite* if for every non-zero vector x, $x^tAx > 0$. For every symmetric positive definite matrix, there exists a lower triangular matrix C such that $A = CC^t$. This factorisation is called Cholesky decomposition[2]. Within Sage, it is obtained by the `cholesky()` method. In the following example, we construct a matrix A which is almost surely positive definite:

```
sage: m = random_matrix(RDF, 10)
sage: A = transpose(m)*m
sage: C = A.cholesky()
```

[2]Cholesky (1875-1918) did his studies at the French École Polytechnique and was artillery officer; his method was invented to solve geodesic problems.

It should be noted that it is not possible to easily know (without performing the Cholesky decomposition) whether a symmetric matrix is positive definite; if we apply the `cholesky` method to a matrix that is not positive definite, the decomposition will fail and an exception `ValueError` will be raised.

To solve a system $Ax = b$ with the Cholesky decomposition, we proceed as with the LU decomposition. Once the Cholesky decomposition is computed, we call `A.solve_right(b)`. Here again, the decomposition is not recomputed.

Why use the Cholesky decomposition instead of the LU decomposition to solve linear systems with symmetric positive definite matrix? First, the memory necessary to store the factors is halved, due to symmetry, but the Cholesky method is especially interesting for its number of operations: indeed, for a size-n matrix, the Cholesky factorisation costs n square roots, $n(n-1)/2$ divisions, $(n^3-n)/6$ additions and as many multiplications. As a comparison, the LU factorisation costs $n(n-1)/2$ divisions as well, but $(n^3-n)/3$ additions and as many multiplications.

13.2.5 The QR Decomposition

Let $A \in \mathbb{R}^{n\times m}$, with $n \geq m$. We want to find two matrices Q and R such that $A = QR$ where $Q \in \mathbb{R}^{n\times n}$ is orthogonal ($Q^t \cdot Q = I$) and $R \in \mathbb{R}^{n\times m}$ is upper triangular. Of course, once such a decomposition is computed, we can use it to solve linear systems if the matrix A is square and invertible. However, as we will see, the QR decomposition is especially interesting to solve least squares problems, and to compute eigenvalues. We should note that A is not necessarily square. The QR decomposition always exists. Example:

```
sage: A = random_matrix(RDF,6,5)
sage: Q, R = A.QR()
```

Exercise 47 (Perturbing a linear system)**.** Let A be an invertible square matrix, and assume we have computed a decomposition of A (LU, QR, Cholesky, ...). Let u and v be two vectors. We consider the matrix $B = A + uv^t$, and we assume that $1 + v^t A^{-1} u \neq 0$. How to cheaply solve the $Bx = f$ system (i.e., without a factorisation of B)?

Hint: We will use the Sherman and Morrison formula (that we will either prove or assume):

$$(A + uv^t)^{-1} = A^{-1} - \frac{A^{-1}uv^tA^{-1}}{1 + v^tA^{-1}u}.$$

13.2.6 Singular Value Decomposition

Let A be an $n \times m$ matrix with real coefficients. Then two orthogonal matrices $U \in \mathbb{R}^{n\times n}$ and $V \in \mathbb{R}^{m\times m}$ exist, such that

$$U^tAV = \Sigma = \mathrm{diag}(\sigma_1, \sigma_2, \cdots, \sigma_p),$$

where $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_p \geq 0$ (with $p = \min(m, n)$). The numbers $\sigma_1, \ldots, \sigma_p$ are the *singular values* of A.

The matrices U and V are orthogonal ($U \cdot U^t = I$ and $V \cdot V^t = I$) and as a consequence:

$$A = U\Sigma V^t.$$

Example (the computations are inexact due to rounding errors):

```
sage: A = matrix(RDF, [[1,3,2],[1,2,3],[0,5,2],[1,1,1]])
sage: U, Sig, V = A.SVD()
sage: A1 = A - U*Sig*transpose(V); A1
[ 2.220446049250313e-16                  0.0                   0.0]
[3.3306690738754696e-16 -4.440892098500626e-16 -4.440892098500626e-16]
[-9.298117831235686e-16 1.7763568394002505e-15 -4.440892098500626e-16]
[ 4.440892098500626e-16 -8.881784197001252e-16 -4.440892098500626e-16]
```

We can show that the singular values of a matrix A are the square roots of the eigenvalues of A^tA. It is easy to check that, for a square matrix of order n, the Euclidean norm $\|A\|_2$ equals σ_1 and that, if the matrix is non-singular, the condition number of A in the Euclidean norm equals σ_1/σ_n. The rank of A is the integer r defined by:

$$\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r > \sigma_{r+1} = \cdots = \sigma_p = 0.$$

13.2.7 Application to Least Squares

We want to solve the over-determined system $Ax = b$ where A is a rectangular matrix with real coefficients, having n rows and m columns with $n > m$. Clearly, this system has no solution in general. We thus consider the minimisation problem of the square Euclidean norm $\|\cdot\|_2$ of the residue:

$$\min_x \|Ax - b\|_2^2.$$

The rank of the matrix A might even be less than m.

Solving the Normal Equations. It is straightforward that the solution satisfies:

$$A^tAx = A^tb.$$

Assuming A of full rank m, we can thus try to form the matrix A^tA and solve the system $A^tAx = A^tb$, for example by computing the Cholesky decomposition of A^tA. This is precisely the origin of Cholesky's method. What is the condition number of A^tA? This is what will influence the accuracy of the results. The singular values of A^tA, which is of dimension $m \times m$, are the squares of the singular values of A; thus the condition number in Euclidean norm is σ_1^2/σ_m^2, the square of the condition number of A, which can be large. We thus prefer the methods based either on the QR decomposition of A, or on its singular value decomposition.

Nevertheless, this method is useful for small systems, with a condition number which is not too bad. Here is the corresponding code:

```
sage: A = matrix(RDF, [[1,3,2],[1,4,2],[0,5,2],[1,3,2]])
sage: b = vector(RDF, [1,2,3,4])
sage: Z = transpose(A)*A
sage: C = Z.cholesky()
sage: R = transpose(A)*b
sage: Z.solve_right(R)
(-1.5000000000000135, -0.5000000000000085, 2.7500000000000213)
```

We should note that Cholesky's decomposition is *cached*, and is used by `Z.solve_right(R)`, without being recomputed.

With the QR decomposition. Assume A of full rank[3]; we consider the QR decomposition of A. Then

$$\|Ax - b\|_2^2 = \|QRx - b\|_2^2 = \|Rx - Q^t b\|_2^2.$$

We have: $R = \begin{bmatrix} R_1 \\ 0 \end{bmatrix}$ where R_1 is $m \times m$ upper triangular and $Q^t b = \begin{bmatrix} c \\ d \end{bmatrix}$ with c of size m. Thus $\|Ax - b\|_2^2 = \|R_1 x - c\|_2^2 + \|d\|_2^2$, and the minimum is obtained by solving the triangular system $R_1 x = c$:

```
sage: A = matrix(RDF, [[1,3,2],[1,4,2],[0,5,2],[1,3,2]])
sage: b = vector(RDF, [1,2,3,4])
sage: Q, R = A.QR()
sage: R1 = R[0:3,0:3]
sage: b1 = transpose(Q)*b
sage: c = b1[0:3]
sage: R1.solve_right(c)
(-1.499999999999999, -0.49999999999999867, 2.749999999999997)
```

Let us compute the condition number of $A^t A$ in infinite norm:

```
sage: Z = A.transpose()*A
sage: Z.norm(Infinity)*(Z^-1).norm(Infinity)
1992.3750000000168
```

The QR method and the method of normal equations give results that agree to within the roundoff level times $\kappa(A^t A)$.

With the Singular Value Decomposition. The singular value decomposition $A = U\Sigma V^t$ also allows us to compute the solution; moreover, we can use it when A is not of full rank. If A is not the zero matrix, Σ has $0 < r \leq m$ positive coefficients σ_i (assumed in decreasing order). We then have:

$$\|Ax - b\|_2^2 = \|U^t A V V^t x - U^t b\|_2^2.$$

Writing $\lambda = V^t x$, and u_i for the columns of U, we have:

$$\|Ax - b\|_2^2 = \sum_{i=1}^{p} (\sigma_i \lambda_i - u_i^t b)^2 + \sum_{i=p+1}^{m} (u_i^t b)^2.$$

[3]We can avoid that condition with a QR method with pivots.

The minimum is thus attained by taking $\lambda_i = (u_i^t b)/\sigma_i$ for $1 \leq i \leq p$ if $\lambda_i \neq 0$, and $\lambda_i = 0$ otherwise. We finally obtain the solution $x = V\lambda$.

Here is the corresponding Sage program:

```
sage: A = matrix(RDF, [[1,3,2],[1,3,2],[0,5,2],[1,3,2]])
sage: B = vector(RDF, [1,2,3,4])
sage: U, Sig, V = A.SVD()
sage: m = A.ncols()
sage: x = vector(RDF, [0]*m)
sage: lamb = vector(RDF, [0]*m)
sage: for i in range(0,m):
....:     s = Sig[i,i]
....:     if s!=0.0:
....:         lamb[i]=U.column(i)*B/s
sage: x = V*lamb; x
(0.2370370370370367, 0.4518518518518521, 0.3703703703703702)
```

Please note that we have chosen here a matrix of rank 2 (we can check with the `A.rank()` command) and thus not of full rank; there are several solutions to the least squares problem, and the above mathematical analysis shows that x is the solution with the smallest Euclidean norm.

Let us look at the singular values:

```
sage: m = 3; [ Sig[i,i] for i in range(0,m) ]
[8.309316833256451, 1.3983038884881154, 0.0]
```

The rank of the matrix A being 2, the third singular value is necessarily 0.

Example. Among the marvelous applications of the singular value decomposition (SVD), here is a problem (known as the *orthogonal Procrustes problem*) which is hard to solve by another method: let A and $B \in \mathbb{R}^{n\times m}$ be the results of an experiment repeated twice. We wonder if B can be *twisted* to A, i.e., if there exists an orthogonal matrix Q such that $A = BQ$. Naturally, the measures A and B contain some noise, whence the problem has no solution in general. We therefore consider the corresponding least squares problem; we are looking for the orthogonal matrix Q which minimises the square of the Frobenius norm:

$$\|A - BQ\|_F^2.$$

We remember that $\|A\|_F^2 = \operatorname{trace}(A^t A)$. Then

$$\|A - BQ\|_F^2 = \operatorname{trace}(A^t A) + \operatorname{trace}(B^t B) - 2\operatorname{trace}(Q^t B^t A) \geq 0,$$

and we have to maximise $\operatorname{trace}(Q^t B^t A)$. We then compute the SVD of $B^t A$: we have $U^t(B^t A)V = \Sigma$. Let us denote by σ_i the singular values, and $Z = V^t Q^t U$. This matrix is orthogonal, and thus all its coefficients are less or equal to 1 in absolute value. Then:

$$\operatorname{trace}(Q^t B^t A) = \operatorname{trace}(Q^t U \Sigma V^t) = \operatorname{trace}(Z\Sigma) = \sum_{i=1}^{m} Z_{ii}\sigma_i \leq \sum_{i=1}^{m} \sigma_i,$$

and the maximum is obtained for $Q = UV^t$.

```
sage: A = matrix(RDF, [[1,2],[3,4],[5,6],[7,8]])
```

B is obtained by adding a random noise to A, and then applying a rotation R:

```
sage: th = 0.7
sage: R = matrix(RDF, [[cos(th),sin(th)],[-sin(th),cos(th)]])
sage: B = (A + 0.1*random_matrix(RDF,4,2)) * transpose(R)
```

```
sage: C = transpose(B)*A
sage: U, Sigma, V = C.SVD()
sage: Q = U*transpose(V)
```

The random noise is small, and Q is close to R as expected:

```
sage: Q
[ 0.7612151656410958  0.6484993998439783]
[-0.6484993998439782  0.7612151656410955]
sage: R
[0.7648421872844885  0.644217687237691]
[-0.644217687237691 0.7648421872844885]
```

Exercise 48 (Square root of a symmetric semi-definite positive matrix)**.** Let A be a symmetric semi-definite positive matrix (i.e., which satisfies $x^tAx \geq 0$ for any vector x). Show that we can compute a matrix X, also symmetric semi-definite positive, such that $X^2 = A$.

13.2.8 Eigenvalues, Eigenvectors

So far, we have used some direct methods (LU, QR, or Cholesky decompositions), which give a solution in a finite number of operations (the four basic arithmetic operations, and the square root for the Cholesky decomposition). This *cannot hold* for the computation of eigenvalues: indeed (cf. page 294), we can associate to every polynomial a matrix whose eigenvalues are the roots of the polynomial, and we know there is no explicit formula for the roots of a polynomial of degree 5 or more, a formula that a direct method would yield. Also, constructing the characteristic polynomial to compute its roots would be extremely costly (the Faddeev-Le Verrier algorithm allows us to compute the characteristic polynomial of a size-n matrix in $O(n^4)$ operations, which is still considered too expensive). The numerical methods used to compute eigenvalues and eigenvectors are all iterative. Recall also that the singular values of a matrix A are the square roots of the eigenvalues of A^tA: consequently there is no direct method to compute the singular value decomposition.

We will thus build sequences converging towards the eigenvalues (and eigenvectors), and stop the iterations when close enough to the solution[4].

[4]In the examples below, we choose a fixed number of iterations, for simplicity.

The Power Method. Let $A \in \mathbb{C}^{n\times n}$. We choose any norm $\|\cdot\|$ on $\mathbb{C}^n$. Starting from x_0, we consider the vector sequence x_k defined by:

$$x_{k+1} = \frac{Ax_k}{\|Ax_k\|}.$$

If the eigenvalues satisfy $|\lambda_1| > |\lambda_2| > \cdots > |\lambda_n|$, then the sequence x_k converges towards an eigenvector associated to the dominant eigenvalue λ_1. Moreover, the sequence $\nu_k = x_{k+1}^* x_k$ converges towards $|\lambda_1|$. (The assumption that the eigenvalues have different absolute values can be relaxed.)

```
sage: n = 10
sage: A = random_matrix(RDF,n)
sage: A = A*transpose(A) + diagonal_matrix([-RDF(i) for i in [1..n]])
sage: # A satisfies (almost surely) the hypotheses.
sage: X = vector(RDF, [1 for i in range(0,n)])
sage: lam_old = 0
sage: for i in range(0,100):
....:     Z = A*X
....:     X = Z/Z.norm()
....:     lam =  X*A*X
....:     s = abs(lam - lam_old)
....:     print("{i} s={s} lambda={lam}".format(i=i, s=s, lam=lam))
....:     lam_old = lam
....:     if s < 1.e-10:
....:         break
0 s=2.0567754429 lambda=-2.0567754429
1 s=1.25099951088 lambda=-3.30777495378
2 s=1.09226722941 lambda=-4.40004218319
3 s=0.908251200751 lambda=-5.30829338394
4 s=0.721759096603 lambda=-6.03005248054
...
96 s=1.69451061005e-06 lambda=-8.01819929979
97 s=1.54755954895e-06 lambda=-8.01820084735
98 s=1.41335122805e-06 lambda=-8.0182022607
99 s=1.29078087419e-06 lambda=-8.01820355148
```

Now, let us compute:

```
sage: A.eigenvalues()
[6.587670892594506,
 2.693595372897774,
 -8.018217143995244,
 -7.662621462625094,
 -6.107162561729528,
 1.2180921545902392,
 0.11456793007905457,
 -0.30200998716052146,
```

```
-2.3272481190401657,
-3.52779132563669]
```

We have indeed determined the dominant eigenvalue.

This method might look of little interest, but it will appear again for sparse matrices. It also introduces what follows, which is very useful.

The Inverse Power Method. We assume a known approximation μ of an eigenvalue λ_j (with $\mu, \lambda_j \in \mathbb{C}$). How can we determine an eigenvector associated to λ_j?

We assume that $\forall k \neq j,\ 0 < |\mu - \lambda_j| < |\mu - \lambda_k|$, and thus, λ_j is a simple eigenvalue. We then consider $(A - \mu I)^{-1}$, whose largest eigenvalue is $(\lambda_j - \mu)^{-1}$, and we apply the power method to this matrix.

Let us take for example:

```
sage: A = matrix(RDF, [[1,3,2],[1,2,3],[0,5,2]])
```

By calling the method `A.eigenvalues()`, we find the eigenvalues (rounded to 5 significant digits) `6.3929, 0.56052, -1.9535`. We will search the eigenvector associated to the second eigenvalue, starting from an approximation:

```
sage: A = matrix(RDF,[[1,3,2],[1,2,3],[0,5,2]])
sage: mu = 0.56
sage: AT = A - mu*identity_matrix(RDF,3)
sage: X = vector(RDF,[1 for i in range(0,A.nrows())])
sage: lam_old = 0
sage: for i in range(1,1000):
....:     Z = AT.solve_right(X)
....:     X = Z/Z.norm()
....:     lam = X.dot_product(A*X)
....:     s = abs(lam - lam_old)
....:     print("{i} s={s} lambda={lam}".format(i=i, s=s, lam=lam))
....:     lam_old = lam
....:     if s<1.e-10:
....:         break
1 s=0.56423627407 lambda=0.56423627407
2 s=0.00371649959176 lambda=0.560519774478
3 s=2.9833340176e-07 lambda=0.560519476145
4 s=3.30288019157e-11 lambda=0.560519476112
sage: X
(0.9276845629439007, 0.10329475725387141, -0.3587917847435305)
```

Let us verify that we have calculated an approximation of the selected eigenvalue and of an associated eigenvector:

```
sage: A*X-lam*X
(2.886579864025407e-15, 1.672273430841642e-15, 8.326672684688674e-15)
```

Several remarks can be made:

- we do not compute the inverse of the matrix $A - \mu I$, but we use its LU factorisation, which is computed only once (by the first `solve_right` call);
- we use the iterations to improve an estimation of the selected eigenvalue;
- the convergence is very fast; we can indeed show that (modulo the above hypotheses, and the choice of an initial vector non-orthogonal to the eigenvector q_j associated to λ_j), we have, for the iterates $x^{(i)}$ and $\lambda^{(i)}$:

$$\|x^{(i)} - q_j\| = O\left(\left|\frac{\mu - \lambda_j}{\mu - \lambda_K}\right|^i\right)$$

and

$$\|\lambda^{(i)} - \lambda_j\| = O\left(\left|\frac{\mu - \lambda_j}{\mu - \lambda_K}\right|^{2i}\right),$$

where λ_K is the second closest eigenvalue to μ;

- the condition number of $A - \mu I$ (bounded from below by the ratio between the largest and smallest eigenvalues of $A - \mu I$) is large thus bad; however, we can show that errors are after all not that important!

The QR Algorithm. Let A be a non-singular square matrix. We consider the sequence $A_0 = A$, A_1, A_2, ..., A_k, A_{k+1}, ... In the raw form of the QR algorithm, going from A_k to A_{k+1} is done as follows:

1. we compute the QR decomposition of A_k: $A_k = Q_k R_k$,
2. we compute $A_{k+1} = R_k Q_k$.

Let us program this method with A a symmetric real matrix:

```
sage: m = matrix(RDF, [[1,2,3,4],[1,0,2,6],[1,8,4,-2],[1,5,-10,-20]])
sage: Aref = transpose(m)*m
sage: A = copy(Aref)
sage: for i in range(0,20):
....:     Q, R = A.QR()
....:     A = R*Q
....:     print(A.str(lambda x: RR(x).str(digits=8)))
[    347.58031    -222.89331    -108.24117  -0.067928252]
[   -222.89331     243.51949     140.96827   0.081743964]
[   -108.24117     140.96827     90.867499 -0.0017822044]
[ -0.067928252   0.081743964 -0.0017822044   0.032699348]
...
[     585.03056 -3.2281469e-13 -6.8752767e-14 -9.9357605e-14]
[-3.0404094e-13       92.914265 -2.5444701e-14 -3.3835458e-15]
[-1.5340786e-39  7.0477800e-25       4.0229095  2.7461301e-14]
[ 1.1581440e-82 -4.1761905e-68  6.1677425e-42    0.032266909]
```

We observe a convergence towards an almost diagonal matrix. The diagonal coefficients are approximations to the eigenvalues of A. Let us check:

```
sage: Aref.eigenvalues()
[585.0305586200212, 92.91426499150643, 0.03226690899408103,
    4.022909479477674]
```

We can prove the convergence if the matrix is Hermitian positive definite. If we have a non-symmetric matrix, we should compute in $\mathbb{C}$, the eigenvalues being *a priori* complex, and, if the method converges, the lower triangular part of A_k tends to zero, while the diagonal tends to the eigenvalues of A.

The QR method requires several improvements to become efficient, in particular because the successive QR decompositions are expensive; among the common refinements, in general we first reduce the matrix A to a simpler form (Hessenberg's form: upper triangular plus the first subdiagonal), which makes the QR decompositions much cheaper; then, to speed-up convergence, we apply cleverly chosen translations $A := A + \sigma I$ (see for example [GVL12]). This is precisely the method used by Sage for dense matrices `CDF` or `RDF`.

In Practice. The above programs are mainly given as pedagogical examples; in practice, we will use the methods provided by Sage, which, whenever possible, call optimised routines from the Lapack library. The interface allows either to get only the eigenvalues, or both the eigenvalues and the corresponding eigenvectors:

```
sage: A = matrix(RDF, [[1,3,2],[1,2,3],[0,5,2]])
sage: eigen_vals, eigen_vects = A.eigenmatrix_right()
sage: eigen_vals
[   6.39294791648918                0.0                 0.0]
[                0.0   0.560519476111939                 0.0]
[                0.0                0.0 -1.9534673926011215]
sage: eigen_vects
[ 0.5424840601106511  0.9276845629439008 0.09834254667424457]
[ 0.5544692861094349 0.10329475725386986  -0.617227053099068]
[ 0.6310902116870117 -0.3587917847435306   0.780614827194734]
```

This example computes the diagonal matrix with eigenvalues, and the eigenvector matrix (whose columns correspond to eigenvectors).

Example (Computing the roots of a polynomial). Given a monic polynomial (with real or complex coefficients) $p(x) = x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$, it is easy to check that the eigenvalues of the companion matrix M, defined by $M_{i+1,i} = 1$ and $M_{i,n-1} = -a_i$, are the roots of p (see §8.2.3), which thus gives a method to approximate these roots:

```
sage: def pol2companion(p):
....:     n = len(p)
....:     m = matrix(RDF,n)
....:     for i in range(1,n):
....:         m[i,i-1]=1
....:     m.set_column(n-1,-p)
....:     return m
```

```
sage: q = vector(RDF,[1,-1,2,3,5,-1,10,11])
sage: comp = pol2companion(q); comp
[  0.0   0.0   0.0   0.0   0.0   0.0   0.0  -1.0]
[  1.0   0.0   0.0   0.0   0.0   0.0   0.0   1.0]
[  0.0   1.0   0.0   0.0   0.0   0.0   0.0  -2.0]
[  0.0   0.0   1.0   0.0   0.0   0.0   0.0  -3.0]
[  0.0   0.0   0.0   1.0   0.0   0.0   0.0  -5.0]
[  0.0   0.0   0.0   0.0   1.0   0.0   0.0   1.0]
[  0.0   0.0   0.0   0.0   0.0   1.0   0.0 -10.0]
[  0.0   0.0   0.0   0.0   0.0   0.0   1.0 -11.0]
sage: roots = comp.eigenvalues(); roots
[0.3475215101190289 + 0.5665505533984981*I, 0.3475215101190289 -
    0.5665505533984981*I,
 0.34502377696179265 + 0.43990870238588275*I, 0.34502377696179265 -
     0.43990870238588275*I,
 -0.5172576143252197 + 0.5129582067889322*I, -0.5172576143252197 -
     0.5129582067889322*I,
 -1.3669971645459291, -9.983578180965276]
```

In this example, the polynomial is represented by the list `q` of its coefficients, from 0 to $n-1$. The polynomial $x^2 - 3x + 2$ would thus be represented by `q=[2,-3]`.

13.2.9 Polynomial Curve Fitting: the Devil is Back

Continuous Version. We would like to approximate the function $f(x)$ by a polynomial $P(x)$ of degree $\leq n$, on the interval $[\alpha, \beta]$. We formulate the least squares problem:

$$\min_{a_0,\ldots,a_n \in \mathbb{R}} J(a_0, \ldots, a_n) = \int_\alpha^\beta (f(x) - \sum_{i=0}^{n} a_i x^i)^2 \, \mathrm{d}x.$$

By differentiating $J(a_0, \ldots, a_n)$ with respect to the coefficients a_i, we find that $a_0, \ldots, a_n$ are solutions of the linear system $Ma = F$ where $M_{i,j} = \int_\alpha^\beta x^i x^j \, dx$ and $F_j = \int_\alpha^\beta x^j f(x) \, dx$. We immediately see by looking at the case $\alpha = 0, \beta = 1$ that M is a Hilbert matrix! However a remedy exists: it suffices to use a basis of orthogonal polynomials (for example, for $\alpha = -1$ and $\beta = 1$ we obtain Legendre polynomials); then M becomes the identity matrix.

Discrete Version. We consider m observations $y_1, \ldots, y_m$ of a phenomenon at points $x_1, \ldots, x_m$. We want to fit a polynomial $\sum_{i=0}^{n-1} a_i x^i$ of degree (at most) $n-1$ among those values, with $n \leq m$. We thus minimise the functional:

$$J(a_0, \ldots, a_{n-1}) = \sum_{j=1}^{m} (\sum_{i=0}^{n-1} a_i x_j^i - y_j)^2.$$

Written like this, the problem will produce a matrix close to a Hilbert matrix and the system will be hard to solve accurately. Yet, we notice that $\langle P, Q\rangle = \sum_{j=1}^{m} P(x_j)\cdot Q(x_j)$ defines a scalar product on polynomials of degree at most $n-1$. We can thus first construct n polynomials of increasing degrees, orthonormal for this scalar product, and then diagonalise the linear system. Remembering[5] that the Gram-Schmidt process reduces to a 3-term recurrence for the computation of orthogonal polynomials, we are looking for the polynomial $P_{n+1}(x)$ under the form $P_{n+1}(x) = xP_n(x) - \alpha_n P_{n-1}(x) - \beta_n P_{n-2}(x)$: the `orthopoly` procedure below performs this computation (we represent here polynomials by the list of their coefficients: for example `[1,-2,3]` corresponds to the polynomial $1 - 2x + 3x^2$).

Horner's scheme to evaluate a polynomial yields the following program:

```
sage: def eval(P,x):
....:     if len(P) == 0:
....:         return 0
....:     else:
....:         return P[0]+x*eval(P[1:],x)
```

We can then encode the scalar product of two polynomials:

```
sage: def pscal(P,Q,lx):
....:     return float(sum(eval(P,s)*eval(Q,s) for s in lx))
```

and the operation $P \leftarrow P + aQ$ for two polynomials P and Q:

```
sage: def padd(P,a,Q):
....:     for i in range(0,len(Q)):
....:         P[i] += a*Q[i]
```

A more careful program should raise an exception when wrongly used; in our case, we use the following exception when $n > m$:

```
sage: class BadParamsforOrthop(Exception):
....:     def __init__(self, degreeplusone, npoints):
....:         self.deg = degreeplusone - 1
....:         self.np = npoints
....:     def __str__(self):
....:         return "degree: " + str(self.deg) + \
....:                " nb. points: " + repr(self.np)
```

The following procedure computes the n orthogonal polynomials:

```
sage: def orthopoly(n,x):
....:     if n > len(x):
....:         raise BadParamsforOrthop(n, len(x))
....:     orth = [[1./sqrt(float(len(x)))]]
....:     for p in range(1,n):
....:         nextp = copy(orth[p-1])
....:         nextp.insert(0,0)
```

[5] Proving it is not very difficult!

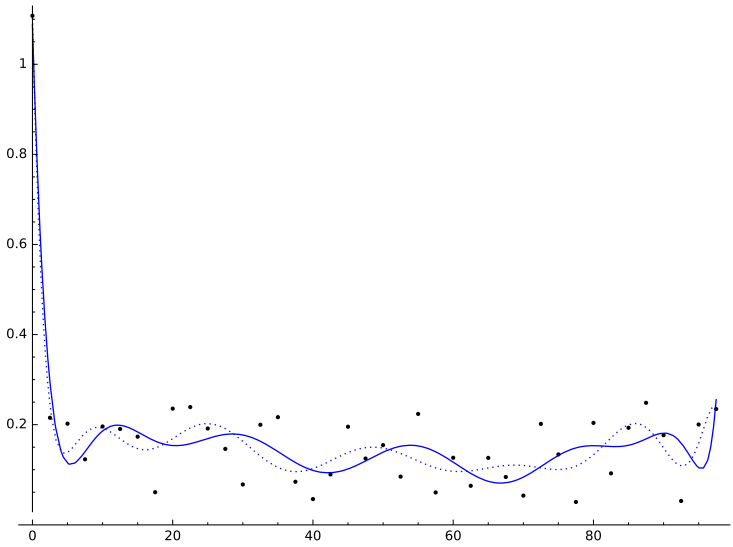

FIGURE 13.1 – Dotted line: naive fitting. Continuous line: orthogonal polynomials fitting.

```
....:         s = []
....:         for i in range(p-1,max(p-3,-1),-1):
....:             s.append(pscal(nextp, orth[i], x))
....:         j = 0
....:         for i in range(p-1,max(p-3,-1),-1):
....:             padd(nextp, -s[j], orth[i])
....:             j += 1
....:         norm = sqrt(pscal(nextp, nextp, x))
....:         nextpn = [nextp[i]/norm for i in range(len(nextp))]
....:         orth.append(nextpn)
....:     return orth
```

Once the orthogonal polynomials $P_0(x), \ldots, P_{n-1}(x)$ are computed, the solution is given by $P(x) = \sum_{i=0}^{n-1} \gamma_i P_i(x)$, with:

$$\gamma_i = \sum_{j=1}^{m} P_i(x_j) y_j,$$

which can be clearly expressed in turn in the monomial basis $1, x, \ldots, x^{n-1}$.

Example ($n = 15$):

```
sage: L = 40
sage: X = [100*float(i)/L for i in range(40)]
sage: Y = [float(1/(1+25*X[i]^2)+0.25*random()) for i in range(40)]
sage: n = 15; orth = orthopoly(n, X)
```

Let us compute the solution coefficients on the basis of orthogonal polynomials:

```
sage: coeff = [sum(Y[j]*eval(orth[i],X[j]) for j in
....:         range(0,len(X))) for i in range(0,n)]
```

We can then translate this result into the monomial basis $1, x, \ldots, x^{n-1}$, for example to draw the graph:

```
sage: polmin = [0 for i in range(0,n)]
sage: for i in range(0,n):
....:     padd(polmin, coeff[i], orth[i])
sage: p = lambda x: eval(polmin, x)
sage: plot(p(x), x, 0, X[len(X)-1])
```

We do not detail here the computation of the naive fitting on the monomial basis x^i, and its graphical representation. We obtain Figure 13.1. The two curves, which correspond to the fitting with a basis of orthogonal polynomials and to the monomial basis, are close; however, if we compute their residue (the value of the functional J) we find 0.1202 for the orthogonal polynomial basis, and 0.1363 for the naive fitting.

13.2.10 Implementation and Efficiency

The computations with matrices having `RDF` coefficients are performed with the processor floating-point unit, those having `RR` coefficients with the GNU MPFR library. Moreover, in the first case, Sage uses NumPy/SciPy, which in turn calls the Lapack library (written in Fortran), this library using the BLAS[6], which are optimised for each processor. We then get for the product of two matrices of size 1000:

```
sage: a = random_matrix(RR, 1000)
sage: b = random_matrix(RR, 1000)
sage: %time _ = a*b
CPU times: user 7min 50s, sys: 508 ms, total: 7min 50s
Wall time: 7min 51s
```

```
sage: c = random_matrix(RDF, 1000)
sage: d = random_matrix(RDF, 1000)
sage: %time _ = c*d
CPU times: user 40 ms, sys: 4 ms, total: 44 ms
Wall time: 45.2 ms
```

whence a ratio of more than 10000 between the multiplication times! (Recall that we compute with the same precision in both cases).

We can also notice the efficiency of computations with matrices having `RDF` coefficients: since the product of two square matrices of size n costs n^3 multiplications (and as many additions), we perform here 10^9 additions and multiplications in 0.045 second; this is about $44 \cdot 10^9$ floating-point operations per second, which corresponds to 44 gigaflops. The processor clock having a frequency of 3.3 Ghz, we thus perform *more* than one operation by clock cycle: this is possible thanks

[6]Basic Linear Algebra Subroutines (matrix-vector products, matrix-matrix products, etc.).

to the almost direct call to the BLAS corresponding routine (which uses the different possibilities to compute in parallel in modern processors cores). There exists an algorithm of cost lower than n^3 to compute the product of two matrices, namely Strassen's algorithm. It is not often used in practice for floating-point computations, as it is more sensitive to roundoff errors as the naive method[Hig93].

13.3 Sparse Matrices

Sparse matrices arise quite often in scientific computing: the sparsity is indeed a wanted property which enables us to solve problems of large size, out of reach with dense matrices.

An approximate definition: we will say that a sequence M_n of matrices, with M_n of size n, is a family of sparse matrices if the number of non-zero coefficients of M_n is $O(n)$.

Clearly, those matrices are encoded in the computer using data structures where only the non-zero elements are stored. By taking into account the sparsity of the matrices, we want of course to save memory to be able to represent large matrices, but also heavily reduce the computation cost.

13.3.1 Where do Sparse Systems Come From?

Boundary Problems. The most common source of sparse linear systems is the discretisation of partial derivative equations. Let us consider for example the Poisson equation (stationary heat equation):

$$-\Delta u = f$$

where $u = u(x, y)$, $f = f(x, y)$,

$$\Delta u := \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}.$$

The equation is considered in the square $[0, 1]^2$, with boundary conditions $u = 0$ on the square border. The one-dimensional analogue is the problem

$$-\frac{\partial^2 u}{\partial x^2} = f, \tag{13.1}$$

with $u(0) = u(1) = 0$.

To approximate the solution of this equation, one of the simplest methods consists in using the finite difference method: we divide the range $[0, 1]$ into a finite number N of intervals of constant width h. We denote by u_i the approximation of u at the point $x_i = ih$. We approximate the derivative of u by $(u_{i+1} - u_i)/h$, and its second derivative by

$$\frac{(u_{i+1} - u_i)/h - (u_i - u_{i-1})/h}{h} = \frac{u_{i+1} - 2u_i + u_{i-1}}{h^2}.$$

We immediately see that the values $u_0, \ldots, u_N$, approximating u at points ih, satisfy a linear system having only 3 non-zero terms per row (and whose matrix is symmetric positive definite).

In dimension 2, we can stick a grid of step h on the unit square, and we obtain a pentadiagonal system (with $4/h^2$ on the diagonal, and $-1/h^2$ on the first two subdiagonals and on the first two superdiagonals). In dimension 3, if we proceed similarly in a cube, we obtain a system where each row has 7 non-zero coefficients. We therefore indeed get very sparse matrices.

Random Walk on a Large Sparse Graph. We consider a graph where each vertex is linked to a small number of vertices (small with respect to the total number of vertices). For example, we can figure out a graph where the vertices are the internet pages: each page only links to a small number of other pages (which defines the graph edges), but it is clearly a huge graph. A random walk on the graph is defined by a stochastic matrix, i.e., a matrix whose coefficients are reals between 0 and 1, with the sum of coefficients being 1 on each line. We can show that such a matrix A has a dominant eigenvalue equal to 1. The stationary distribution of the random walk is the (left) eigenvector x associated to the dominant eigenvalue, i.e., the vector x satisfying $xA = x$. One of the most dramatic applications is the *Pagerank* algorithm from *Google*, where the vector x is used to balance the search results.

13.3.2 Sparse Matrices in Sage

Sage gives us the chance to work with sparse matrices, by adding `sparse = True` when creating the matrix. It corresponds to a representation as a dictionary (§3.3.9). In addition, computations on large sparse matrices with coefficients in `RDF` or `CDF` are performed by Sage using the SciPy library, which offers its own classes of sparse matrices. In the current situation, there is no interface between the sparse matrices of Sage and those of SciPy. We thus have to directly use the SciPy objects.

The available SciPy classes for sparse matrices are:

- a list-of-lists structure (different however from that used by Sage), which is quite handy to create and modify matrices, the `lil_matrix` class;
- some immutable structures, which store only non-zero coefficients, and which are standard formats in sparse linear algebra (`csr` and `csv` formats).

13.3.3 Solving Linear Systems

For systems of moderate size, we can use a direct method, based on the LU decomposition. We can easily convince ourselves that, in the LU decomposition of a sparse matrix A, the L and U factors usually contain more non-zero terms altogether than A. It is necessary to renumber the unknowns to limit the memory used, as in the SuperLU library used by Sage in a transparent manner:

```
sage: from scipy.sparse.linalg.dsolve import *
```

```
sage: from scipy.sparse import lil_matrix
sage: from numpy import array
sage: n = 200
sage: n2 = n*n
sage: A = lil_matrix((n2, n2))
sage: h2 = 1./float((n+1)^2)
sage: for i in range(0,n2):
....:    A[i,i]=4*h2+1.
....:    if i+1<n2: A[i,int(i+1)]=-h2
....:    if i>0:    A[i,int(i-1)]=-h2
....:    if i+n<n2: A[i,int(i+n)]=-h2
....:    if i-n>=0: A[i,int(i-n)]=-h2
sage: Acsc = A.tocsc()
sage: b = array([1 for i in range(0,n2)])
sage: solve = factorized(Acsc) # LU factorisation
sage: S = solve(b)             # resolution
```

Once we have created the matrix as a `lil_matrix` (warning, this format requires indices of type `int` from Python), we have to convert it to the `csc` format. The above program is not very efficient: the construction of the `lil_matrix` is slow, the `lil_matrix` data structure being quite inefficient. However, the conversion to a `csc` matrix and its factorisation are fast; the following resolution is even faster.

Iterative Methods. The main principle of these methods is to build a sequence converging towards the solution of the $Ax = b$ system. Modern iterative methods rely on the Krylov space K_n, the vector space spanned by $\{b, Ab, \ldots, A^n b\}$. Among the most popular methods, let us mention:

- the conjugate gradient method: it can only be used for systems whose matrix A is symmetric positive definite. In this case $\|x\|_A = \sqrt{x^t Ax}$ is a norm, and the iterate x_n is such that it minimises the error $\|x - x_n\|_A$ between the solution x and x_n for $x_n \in K_n$ (some formulas exist which are easy to program, cf. for example [GVL12]);
- the generalised minimal residual method (GMRES): it is designed for non-symmetric linear systems. At the n-th iteration, the Euclidean residual norm $\|Ax_n - b\|_2$ is minimised for $x_n \in K_n$. We notice that it is a least squares problem.

In practice, these methods are only efficient with *preconditioning*: instead of solving $Ax = b$, we solve $MAx = Mb$ where M is a matrix such that MA has a better condition number than A. The study and discovery of efficient preconditioners is an active branch of numerical analysis, with fruitful developments. As an example, here is the resolution of the system studied above by the conjugate gradient method, where the preconditioner M is the inverse of the diagonal of A. It is a simple but not very efficient preconditioner:

```
sage: b = array([1 for i in range(0,n2)])
```

Most useful commands (dense matrices)	
Solve a linear system	`x = A\b`
LU decomposition	`P, L, U =  A.LU()`
Cholesky decomposition	`C = A.cholesky()`
QR decomposition	`Q, R = A.QR()`
Singular value decomposition	`U, Sig, V = A.SVD()`
Eigenvalues and eigenvectors	`Val, Vect = A.eigenmatrix_right()`

TABLE 13.1 – Summary.

```
sage: m = lil_matrix((n2, n2))
sage: for i in range(0,n2):
....:     m[i,i] = 1./A[i,i]
sage: msc = m.tocsc()
sage: from scipy.sparse.linalg import cg
sage: x = cg(A, b, M = msc, tol=1.e-8)
```

13.3.4 Eigenvalues, Eigenvectors

The Power Method. The power method is particularly well suited for huge sparse matrices; indeed, to implement the algorithm, it is enough to know how to perform matrix-vector and scalar products. As an example, let us come back to random walks on a sparse graph, and let us compute the stationary distribution using the power method:

```
sage: from scipy import sparse
sage: from numpy.linalg import *
sage: from numpy import array
sage: from numpy.random import rand
sage: def power(A,x):              # power iteration
....:     for i in range(0,1000):
....:         y = A*x
....:         z = y/norm(y)
....:         lam = sum(x*y)
....:         s = norm(x-z)
....:         print("{i} s={s} lambda={lam}".format(i=i, s=s, lam=lam))
....:         if s < 1e-3:
....:             break
....:         x = z
....:     return x
sage: n = 1000
sage: m = 5
sage: # build a stochastic matrix of size n
sage: # with m non-zero coefficients per row
sage: A1 = sparse.lil_matrix((n, n))
```

```
sage: for i in range(0,n):
....:     for j in range(0,m):
....:         l = int(n*rand())
....:         A1[l,i] = rand()
sage: for i in range(0,n):
....:     s = sum(A1[i,0:n])
....:     A1[i,0:n] /= s
sage: At = A1.transpose().tocsc()
sage: x = array([rand() for i in range(0,n)])
sage: # compute the dominant eigenvalue
sage: # and the associated eigenvector
sage: y = power(At, x)
0 s=17.0241218112 lambda=235.567796432
1 s=0.39337173784 lambda=0.908668201953
2 s=0.230865716856 lambda=0.967356896036
3 s=0.134156683993 lambda=0.986660315554
4 s=0.0789423487458 lambda=0.995424635219
...
```

When running this example, we might play with measuring the time spent in its different parts, and we will observe that almost all the time is spent in constructing the matrix; computing the transpose is not very expensive; the power iterations themselves take negligible time (about 2% of the total time on the test computer). The representation using list of large matrices is not very efficient, and this kind of problem should be rather solved with compiled languages and appropriate data structures.

13.3.5 More Thoughts on Solving Large Non-Linear Systems

The power method and the methods using the Krylov space share a key property: they only require the computation of matrix-vector products. We do not even need to know the matrix, we only need to know the *action* of the matrix on a vector. This is why these methods are also called "black box" methods. It is thus possible to perform some computations in cases where the matrix is not explicitly known, or when we are unable to compute it. The SciPy methods allow in fact *linear operators* as arguments. We invite the reader to consider the following application, and maybe implement it.

Let $F : \mathbb{R}^n \to \mathbb{R}^n$. We want to solve $F(x) = 0$. We consider Newton's method, where we compute iterates $x_{n+1} = x_n - J(x_n)^{-1} \cdot F(x_n)$, starting from x_0. The Jacobian matrix $J(x_n)$ is the matrix of partial derivatives of F at x_n. In practice, one will successively solve $J(x_n)y_n = F(x_n)$, then compute $x_{n+1} = x_n - y_n$. We thus have to solve a linear system. If F is somewhat hard to compute, then its partial derivatives are in general still harder to compute and to program, and this computation might be almost impossible. We therefore bring automatic differentiation to the rescue: if e_j is the vector from $\mathbb{R}^n$ with all its components 0 except the j-th one equal to 1, then $(F(x + he_j) - F(x))/h$ yields a (good)

approximation of the j-th column of the Jacobian matrix for small h. We thus have to perform $n+1$ evaluations of F to obtain an approximation of J, which is quite expensive if n is large. How about applying an iterative method like Krylov to solve the system $J(x_n)y_n = F(x_n)$? We notice that $J(x_n)V \simeq (F(x_n+hV)-F(x_n))/h$ for h small enough, which avoids the computation of the *whole* matrix. Within SciPy, it is sufficient to define a "linear" operator as being the application $V \to (F(x_n + hV) - F(x_n))/h$. This kind of method is quite often used to solve large non-linear systems. The "matrix" being non-symmetric, we will use for example the GMRES method.