- *Discounted total.* Let c be an n-vector representing a cash flow, with c_i the cash received (when $c_i > 0$) in period i. Let d be the n-vector defined as

$$d = (1, 1/(1+r), \ldots, 1/(1+r)^{n-1}),$$

  where $r \geq 0$ is an interest rate. Then

$$d^T c = c_1 + c_2/(1+r) + \cdots + c_n/(1+r)^{n-1}$$

  is the discounted total of the cash flow, *i.e.*, its *net present value* (NPV), with interest rate r.

- *Portfolio value.* Suppose s is an n-vector representing the holdings in shares of a portfolio of n different assets, with negative values meaning short positions. If p is an n-vector giving the prices of the assets, then $p^T s$ is the total (or net) value of the portfolio.

- *Portfolio return.* Suppose r is the vector of (fractional) returns of n assets over some time period, *i.e.*, the asset relative price changes

$$r_i = \frac{p_i^{\text{final}} - p_i^{\text{initial}}}{p_i^{\text{initial}}}, \quad i = 1, \ldots, n,$$

  where p_i^{initial} and p_i^{final} are the (positive) prices of asset i at the beginning and end of the investment period. If h is an n-vector giving our portfolio, with h_i denoting the dollar value of asset i held, then the inner product $r^T h$ is the total return of the portfolio, in dollars, over the period. If w represents the fractional (dollar) holdings of our portfolio, then $r^T w$ gives the total return of the portfolio. For example, if $r^T w = 0.09$, then our portfolio return is 9%. If we had invested \$10000 initially, we would have earned \$900.

- *Document sentiment analysis.* Suppose the n-vector x represents the histogram of word occurrences in a document, from a dictionary of n words. Each word in the dictionary is assigned to one of three sentiment categories: *Positive*, *Negative*, and *Neutral*. The list of positive words might include 'nice' and 'superb'; the list of negative words might include 'bad' and 'terrible'. Neutral words are those that are neither positive nor negative. We encode the word categories as an n-vector w, with $w_i = 1$ if word i is positive, with $w_i = -1$ if word i is negative, and $w_i = 0$ if word i is neutral. The number $w^T x$ gives a (crude) measure of the sentiment of the document.

1.5 Complexity of vector computations

Computer representation of numbers and vectors. Real numbers are stored in computers using *floating point format*, which represents a real number using a block of 64 *bits* (0s and 1s), or 8 *bytes* (groups of 8 bits). Each of the 2^{64} possible sequences of bits corresponds to a specific real number. The floating point numbers

span a very wide range of values, and are very closely spaced, so numbers that arise in applications can be approximated as floating point numbers to an accuracy of around 10 digits, which is good enough for almost all practical applications. Integers are stored in a more compact format, and are represented exactly.

Vectors are stored as arrays of floating point numbers (or integers, when the entries are all integers). Storing an n-vector requires $8n$ bytes to store. Current memory and storage devices, with capacities measured in many gigabytes (10^9 bytes), can easily store vectors with dimensions in the millions or billions. Sparse vectors are stored in a more efficient way that keeps track of indices and values of the nonzero entries.

Floating point operations. When computers carry out addition, subtraction, multiplication, division, or other arithmetic operations on numbers represented in floating point format, the result is rounded to the nearest floating point number. These operations are called *floating point operations*. The very small error in the computed result is called (floating point) *round-off error*. In most applications, these very small errors have no practical effect. Floating point round-off errors, and methods to mitigate their effect, are studied in a field called *numerical analysis*. In this book we will not consider floating point round-off error, but you should be aware that it exists. For example, when a computer evaluates the left-hand and right-hand sides of a mathematical identity, you should not be surprised if the two numbers are not equal. They should, however, be very close.

Flop counts and complexity. So far we have seen only a few vector operations, like scalar multiplication, vector addition, and the inner product. How quickly these operations can be carried out by a computer depends very much on the computer hardware and software, and the size of the vector.

A very rough estimate of the time required to carry out some computation, such as an inner product, can be found by counting the total number of floating point operations, or FLOPs. This term is in such common use that the acronym is now written in lower case letters, as flops, and the speed with which a computer can carry out flops is expressed in Gflop/s (gigaflops per second, *i.e.*, billions of flops per second). Typical current values are in the range of 1–10 Gflop/s, but this can vary by several orders of magnitude. The actual time it takes a computer to carry out some computation depends on many other factors beyond the total number of flops required, so time estimates based on counting flops are very crude, and are not meant to be more accurate than a factor of ten or so. For this reason, gross approximations (such as ignoring a factor of 2) can be used when counting the flops required in a computation.

The *complexity* of an operation is the number of flops required to carry it out, as a function of the size or sizes of the input to the operation. Usually the complexity is highly simplified, dropping terms that are small or negligible (compared to other terms) when the sizes of the inputs are large. In theoretical computer science, the term 'complexity' is used in a different way, to mean the number of flops of the best method to carry out the computation, *i.e.*, the one that requires the fewest flops. In this book, we use the term complexity to mean the number of flops required by a specific method.

Complexity of vector operations. Scalar-vector multiplication ax, where x is an n-vector, requires n multiplications, *i.e.*, ax_i for $i = 1, \ldots, n$. Vector addition $x+y$ of two n-vectors takes n additions, *i.e.*, $x_i + y_i$ for $i = 1, \ldots, n$. Computing the inner product $x^T y = x_1 y_1 + \cdots + x_n y_n$ of two n-vectors takes $2n-1$ flops, n scalar multiplications and $n-1$ scalar additions. So scalar multiplication, vector addition, and the inner product of n-vectors require n, n, and $2n-1$ flops, respectively. We only need an estimate, so we simplify the last to $2n$ flops, and say that the *complexity* of scalar multiplication, vector addition, and the inner product of n-vectors is n, n, and $2n$ flops, respectively. We can guess that a 1 Gflop/s computer can compute the inner product of two vectors of size one million in around one thousandth of a second, but we should not be surprised if the actual time differs by a factor of 10 from this value.

The *order* of the computation is obtained by ignoring any constant that multiplies a power of the dimension. So we say that the three vector operations scalar multiplication, vector addition, and inner product have order n. Ignoring the factor of 2 dropped in the actual complexity of the inner product is reasonable, since we do not expect flop counts to predict the running time with an accuracy better than a factor of 2. The order is useful in understanding how the time to execute the computation will scale when the size of the operands changes. An order n computation should take around 10 times longer to carry out its computation on an input that is 10 times bigger.

Complexity of sparse vector operations. If x is sparse, then computing ax requires $\mathbf{nnz}(x)$ flops. If x and y are sparse, computing $x+y$ requires no more than $\min\{\mathbf{nnz}(x), \mathbf{nnz}(y)\}$ flops (since no arithmetic operations are required to compute $(x+y)_i$ when either x_i or y_i is zero). If the sparsity patterns of x and y do not overlap (intersect), then zero flops are needed to compute $x+y$. The inner product calculation is similar: computing $x^T y$ requires no more than $2\min\{\mathbf{nnz}(x), \mathbf{nnz}(y)\}$ flops. When the sparsity patterns of x and y do not overlap, computing $x^T y$ requires zero flops, since $x^T y = 0$ in this case.