Introduction to Scientific Computing
Lecture 2

Professor Hanno Rein

Last updated: September 12, 2016

Review of Floating Point Numbers

The basic idea is as follows:

$$x = \text{sign} \times \text{mantissa} \times 2^{\text{exponent}}$$

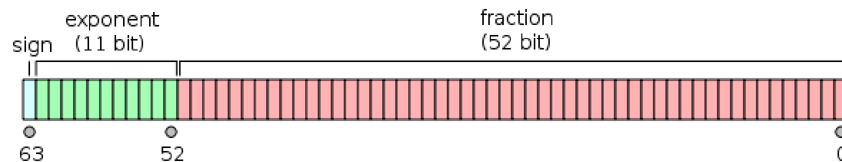

Figure 1: Double floating point number in memory. Source: `http://en.wikipedia.org`.

We work in IEE754 double floating point precision. Which of the following numbers can be represented and if so can they be represented exactly?

```
3.0
2.5
2.4
1e+256
-0.1
1e+512
1e-256
0.5
-0.03125
```

Calculate the following expressions in IEE754

```
1+1=
1e+300 x 1e+100 =
1e+200 - (1e+200 + 1) =
(((((0.125 + 1e-18) + 1e-19) + 1e-18) + 1e-19) +
   1e-18)
```

2 Algorithmic Complexity

Naiive implementation in recursive form:

```
def fib_recursive(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib_recursive(n-1) + fib_recursive
          (n-2)
```

This is exponentially slow! We say that it scales as $O(2^n)$.

What we're trying to here is to characterize an algorithm's efficiency in terms of execution time (also called run-time or wall-time). We want to do this independent of the specific programming language, compiler optimizations or the speed of a specific computer. We will quantify the number of operations or steps that the algorithm requires. This allows us to considered each of these steps as a basic unit of computation, then the execution time for an algorithm can be expressed as the number of steps required to solve the problem. What this basic unit of computation is in the end, is not very important for us right now, we are only interested in the *scaling*.

As the problem gets larger, some portion of out algorithm will take longer. We here focus on the slowest part and ignore all other parts. For example, the calculation of the Fibonacci numbers might take a long time, whereas the printing of the result is very fast. The dominant part is what is used for comparisons. The order of magnitude function describes the part that increases the fastest as the value of n increases. Order of magnitude is often called Big-O notation (for "order") and written as $O(\cdot)$. It provides a useful approximation to the actual number of steps in the computation.

f(n)	Name
1	Constant
$\log n$	Logarithmic
n	Linear
$n \log n$	Log Linear
n^2	Quadratic
n^3	Cubic
2^n	Exponential

Figure 2: Scalings often found in numerical algorithms. Source: http://interactivepython.org.

2.1 Fibonacci Numbers Take 2

The next big things we'll look at are matrix decomposition, eigenvalues and eigenvectors.

But before we go into the meaty stuff, let's look at the Fibonacci numbers again. We wrote the recursion sequence before as $F_n = F_{n-2} + F_{n-1}$ with $F_0 = 0$ and $F_1 = 1$. Let's write this as a matrix equation! You can easily convince yourself that the following equation will also produce the Fibonacci numbers:

$$\begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} F_{n-1} \\ F_{n-2} \end{pmatrix}.$$

We can furthermore introduce a vector and write it as

$$\vec{F}_{n+1} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \cdot \vec{F}_n.$$

The starting value is

$$\vec{F}_1 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}.$$

Now that we have a matrix equation we can look at things like eigenvectors and eigenvalues. It'll become clear why we want to do that in a minute.

I'll give you 5 minutes to calculate the eigenvalues and eigenvectors of

$$A = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

The answers are:

$$\lambda_1 = \frac{1+\sqrt{5}}{2} \qquad \text{with} \qquad \vec{x}_1 = \begin{pmatrix} \frac{1}{2}(1+\sqrt{5}) \\ 1 \end{pmatrix}$$
$$\lambda_2 = \frac{1-\sqrt{5}}{2} \qquad \text{with} \qquad \vec{x}_2 = \begin{pmatrix} \frac{1}{2}(1-\sqrt{5}) \\ 1 \end{pmatrix}.$$

Let's express the initial value as a combination of eigenvectors, which is simply

$$\vec{F}_1 = \frac{1}{\sqrt{5}} \left(\vec{x}_1 - \vec{x}_2 \right).$$

To get the n-th Fibonacci number, we have to apply the matrix A $n-1$ times to $\vec{F}_1$. Now that we have decomposed F_1 into eigenvectors, the result is very easy to write down as each eigenvector just gets multiplied n times with its eigenvalue.

$$\vec{F}_n = \frac{1}{\sqrt{5}} \left(\lambda_1^{n-1} \vec{x}_1 - \lambda_2^{n-1} \vec{x}_2 \right).$$

or if we only look at the top component of the $\vec{F}$ vector,

$$\begin{aligned} F_n &= \frac{1}{\sqrt{5}} \left(\lambda_1^{n-1} \frac{1}{2}(1+\sqrt{5}) - \lambda_2^{n-1} \frac{1}{2}(1-\sqrt{5}) \right). \\ &= \frac{1}{\sqrt{5}} \left((\frac{1}{2}(1+\sqrt{5}))^n - (\frac{1}{2}(1-\sqrt{5}))^n \right). \end{aligned}$$

We now have an anlytic expression for the Fibonacci numbers. However, we need to use complicated functions like the n-th power and the square root to get the Fibonacci numbers with the above algorithm. Especially in floating point precission, this can be inaccurate.

We can come up with yet another algorithm in $O(\log n)$ instead of $O(n)$. We can proof by induction, that the following is true:

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}$$

Recall that for matricies

$$A^{2n} = A^n A^n.$$

This gives us the relations

$$\begin{aligned} F_{2n-1} &= F_n^2 + F_{n-1}^2 \\ F_{2n} &= (F_{n-1} + F_{n+1})F_n = (2F_{n-1} + F_n)F_n. \end{aligned}$$

Now, to calculate the Fibonacci numbers for large n we can choose either the first equation (for odd numbers) or the second (for even numbers). We need to calculate two new Fibonacci numbers, but they are half the size of the original number. Repeating the process recursively, gives us a very fast algorithm to calculate large number: it is $O(\log n)$ Here is the python algorithm to do that.

```
def fib(n):
    if n==0:
        return 0
    if n==1:
        return 1
    if n%2==0:
        fn = fib(n/2)
        fn1 = fib(n/2-1)
        return (2*fn1+fn)*fn
    if n%2==1:
        fn = fib((n+1)/2)
        fn1 = fib((n+1)/2-1)
        return fn*fn+fn1*fn1
```

Code 1: Recursive python algorithm to calculate Fibonacci numbers.

3 Matrix operations

3.1 Linear equation systems

In the following sections, we will spend some time to solve linear systems of equations. This is a tool that will come in handy in many different places during this course. For some problems, there are specialized algorithms available. For example, matrices can be diagonal, sparse or tri-diagonal. In these cases, optimized algorithms can speed up the calculation dramatically. In our course, we discuss a general purpose algorithm that can be used to do many tasks related to matrices. The most important task is to solve the system of linear equations

$$A \cdot x = b$$

for x. Here, A is a $N \times N$ matrix and x and b are vectors of lengths N. You might have heard of at least one way to solve this system in a linear algebra course: Gaussian elimination. This would also work on a computer. However, we can do even better. We'll implement what is called the LU-decomposition with the Crout algorithm. But before we go into the algorithm itself, let's start with an real world problem where linear equation systems appear: fitting.