- Kahan [19] mentions some strange behavior of some versions of the Excel spreadsheet. They seem to be due to an attempt to mimic a decimal arithmetic with an underlying binary one.

  An even more striking behavior happens with some early versions of Excel 2007: When you try to compute

  $$65536 - 2^{-37}$$

  the displayed result is 100001. This is an error in the binary-to-decimal conversion used for displaying that result: the internal binary value is correct, if you add 1 to that result you get 65537. An explanation can be found at `http://blogs.msdn.com/excel/archive/2007/09/25/calculation-issue-update.aspx`, and a patch is available from `http://blogs.msdn.com/excel/archive/2007/10/09/calculation-issue-update-fix-available.aspx`

- Some bugs do not require any programming error: they are due to poor specifications. For instance, the Mars Climate Orbiter probe crashed on Mars in September 1999 because of an astonishing mistake: one of the teams that designed the numerical software assumed the unit of distance was the meter, while another team assumed it was the foot [1, 32].

  Very similarly, in June 1985, a space shuttle positioned itself to receive a laser beamed from the top of a mountain that was supposedly 10,000 miles high, instead of the correct 10,000 feet [1].

1.3.2 Difficult problems

Sometimes, even with a correctly implemented floating-point arithmetic, the result of a computation is far from what could be expected.

A sequence that seems to converge to a wrong limit

Consider the following example, due to one of us [28] and analyzed by Kahan [19, 30]. Let (u_n) be the sequence defined as

$$\begin{cases} u_0 &= 2 \\ u_1 &= -4 \\ u_n &= 111 - \dfrac{1130}{u_{n-1}} + \dfrac{3000}{u_{n-1}u_{n-2}} \end{cases} . \tag{1.1}$$

One can easily show that the limit of this sequence is 6. And yet, on any system with any precision, the sequence will seem to go to 100.

For example, Table 1.1 gives the results obtained by compiling Program 1.1 and running it on a Pentium4-based workstation, using the GNU Compiler Collection (GCC) and the Linux system.

```
#include <stdio.h>

int main(void)
{
  double u, v, w;
  int i, max;

  printf("n =");
  scanf("%d",&max);
  printf("u0 = ");
  scanf("%lf",&u);
  printf("u1 = ");
  scanf("%lf",&v);
  printf("Computation from 3 to n:\n");
  for (i = 3; i <= max; i++)
    {
      w = 111. - 1130./v + 3000./(v*u);
      u = v;
      v = w;
      printf("u%d = %1.17g\n", i, v);
    }
  return 0;
}
```

Program 1.1: A C program that is supposed to compute sequence u_n using double-precision arithmetic. The obtained results are given in Table 1.1.

The explanation of this weird phenomenon is quite simple. The general solution for the recurrence

$$u_n = 111 - \frac{1130}{u_{n-1}} + \frac{3000}{u_{n-1}u_{n-2}}$$

is

$$u_n = \frac{\alpha \cdot 100^{n+1} + \beta \cdot 6^{n+1} + \gamma \cdot 5^{n+1}}{\alpha \cdot 100^{n} + \beta \cdot 6^{n} + \gamma \cdot 5^{n}},$$

where α, β, and γ depend on the initial values u_0 and u_1. Therefore, if $\alpha \neq 0$ then the limit of the sequence is 100, otherwise (assuming $\beta \neq 0$), it is 6. In the present example, the starting values $u_0 = 2$ and $u_1 = -4$ were chosen so that $\alpha = 0$, $\beta = -3$, and $\gamma = 4$. Therefore, the "exact" limit of u_n is 6. And yet, when computing the values u_n in floating-point arithmetic using (1.1), due to the various rounding errors, even the very first computed terms become slightly different from the exact terms. Hence, the value α corresponding to these computed terms is very tiny, but nonzero. This suffices to make the computed sequence "converge" to 100.