Solution to Exercise 42.

a) `6 / 4 * 2.0f - 3` $\longrightarrow$
`1 * 2.0f - 3` $\longrightarrow$
`2.0f - 3` $\longrightarrow$
`-1.0f`

b) `2 + 15.0e7f - 3 / 2.0 * 1.0e8` $\longrightarrow^{1.5 \cdot 10^8 > 2^{27}}$
`15.0e7f - 3 / 2.0 * 1.0e8` $\longrightarrow$
`15.0e7f - 1.5 * 1.0e8` $\longrightarrow$
`15.0e7f - 1.5e8` $\longrightarrow$
`0.0`

c) `392593 * 2735.0f - 8192 * 131072 + 1.0` $\longrightarrow$
binary: `1`$\underbrace{\texttt{0...0}}_{\text{23 times}}\underbrace{\texttt{0011111}}_{\text{get lost}}$`f - 8192 * 131072 + 1.0` $\longrightarrow$
`1073741824.0f - 8192 * 131072 + 1.0` $\longrightarrow$
`1073741824.0f - 1073741824 + 1.0` $\longrightarrow$
`0.0f + 1.0` $\longrightarrow$
`1.0`

d) `16 * (0.2f + 262144 - 262144.0)` $\longrightarrow$
`16 * (`binary:`1`$\underbrace{\texttt{0...0}}_{\text{18 times}}$`.00110`$\underbrace{\overline{\texttt{0110}}}_{\text{get lost}}$`f - 262144.0)` $\longrightarrow$
`16 *` binary:`0.0011` $\longrightarrow$
`3.0`

Solution to Exercise 43.

a) This is easy and doesn't take any calculations: $0.25 = 1/4 = 1 \cdot 2^{-2}$. As a binary number, this is 0.01.

b) We employ the rules from Section 2.5.5.

$$
\begin{array}{rcrcrcl}
 & & & & 1.52 & \to & b_0 = 1 \\
2(1.52-1) & = & 2\cdot 0.52 & = & 1.04 & \to & b_{-1} = 1 \\
2(1.04-1) & = & 2\cdot 0.04 & = & 0.08 & \to & b_{-2} = 0 \\
2(0.08-0) & = & 2\cdot 0.08 & = & 0.16 & \to & b_{-3} = 0 \\
2(0.16-0) & = & 2\cdot 0.16 & = & 0.32 & \to & b_{-4} = 0 \\
2(0.32-0) & = & 2\cdot 0.32 & = & 0.64 & \to & b_{-5} = 0 \\
2(0.64-0) & = & 2\cdot 0.64 & = & 1.28 & \to & b_{-6} = 1 \\
2(1.28-1) & = & 2\cdot 0.28 & = & 0.56 & \to & b_{-7} = 0 \\
2(0.56-0) & = & 2\cdot 0.56 & = & 1.12 & \to & b_{-8} = 1 \\
2(1.12-1) & = & 2\cdot 0.12 & = & 0.24 & \to & b_{-9} = 0 \\
2(0.24-0) & = & 2\cdot 0.24 & = & 0.48 & \to & b_{-10} = 0 \\
2(0.48-0) & = & 2\cdot 0.48 & = & 0.96 & \to & b_{-11} = 0 \\
2(0.96-0) & = & 2\cdot 0.96 & = & 1.92 & \to & b_{-12} = 1 \\
2(1.92-1) & = & 2\cdot 0.92 & = & 1.84 & \to & b_{-13} = 1 \\
2(1.84-1) & = & 2\cdot 0.84 & = & 1.68 & \to & b_{-14} = 1 \\
2(1.68-1) & = & 2\cdot 0.68 & = & 1.36 & \to & b_{-15} = 1 \\
2(1.36-1) & = & 2\cdot 0.36 & = & 0.72 & \to & b_{-16} = 0 \\
2(0.72-0) & = & 2\cdot 0.72 & = & 1.44 & \to & b_{-17} = 1 \\
2(1.44-1) & = & 2\cdot 0.44 & = & 0.88 & \to & b_{-18} = 0 \\
2(0.88-0) & = & 2\cdot 0.88 & = & 1.76 & \to & b_{-19} = 1 \\
2(1.76-1) & = & 2\cdot 0.76 & = & 1.52 & \to & b_{-20} = 1 \\
 & & & & \vdots & &
\end{array}
$$

Phew, finally the sequence becomes periodic, and we get the binary expansion $1.\overline{10000101000111101011}$.

c) We employ the rules from Section 2.5.5.

$$
\begin{array}{rcrcrcl}
 & & & & 1.3 & \to & b_0 = 1 \\
2(1.3-1) & = & 2\cdot 0.3 & = & 0.6 & \to & b_{-1} = 0 \\
2(0.6-0) & = & 2\cdot 0.6 & = & 1.2 & \to & b_{-2} = 1 \\
2(1.2-1) & = & 2\cdot 0.2 & = & 0.4 & \to & b_{-3} = 0 \\
2(0.4-0) & = & 2\cdot 0.4 & = & 0.8 & \to & b_{-4} = 0 \\
2(0.8-0) & = & 2\cdot 0.8 & = & 1.6 & \to & b_{-5} = 1 \\
2(1.6-1) & = & 2\cdot 0.6 & = & 1.2 & \to & b_{-6} = 1 \\
 & & & & \vdots & &
\end{array}
$$

We see that the expansion is periodic and yields the binary number $1.0\overline{1001}$.

d) We write $11.1 = 10 + 1.1$ and add the binary expansion 1010.0 of 10 to the binary expansion $1.0\overline{0011}$ of 1.1 derived in Section 2.5.5. The resulting expansion is $1011.0\overline{0011}$.

Solution to Exercise 44.

a) 0.25 has normalized binary floating point representation $1.0 \cdot 2^{-2}$ and is therefore smaller than any number in $\mathcal{F}^*(2,5,-1,2)$. The nearest number is therefore the smallest number in this system, namely 0.5 with normalized binary representation $1.0 \cdot 2^{-1}$. In $\mathcal{F}(2,5,-1,2)$, we can represent 0.25 exactly as $0.1 \cdot 2^{-1}$.

b) 1.52 has normalized binary floating point representation $1.\overline{10000101000111101011} \cdot 2^0$. To get the nearest number in $\mathcal{F}^*(2,5,-1,2)$, we have to round to 5 significant digits. The result is $1.1000 \cdot 2^0 = 1.5$, obtained by rounding down, since $1.1001 \cdot 2^0 = 1.5625$, obtained by rounding up, is farther away. The nearest number in $\mathcal{F}(2,5,-1,2)$ is the same, since this system has only extra numbers *smaller* than any normalized number. Such numbers cannot be nearest to numbers *larger* than some normalized number.

c) 1.3 has normalized binary floating point representation $1.0\overline{1001}$. To get the nearest number in $\mathcal{F}^*(2,5,-1,2)$, we have to round to 5 significant digits. The result is $1.0101 \cdot 2^0 = 1.3125$, obtained from rounding up, since $1.0100 \cdot 2^0 = 1.25$, obtained from rounding down, is farther away. The nearest number in $\mathcal{F}(2,5,-1,2)$ is the same.

d) 11.1 is larger than any number in the system $\mathcal{F}^*(2,5,-1,2)$. Recall that the largest number is $1.1111 \cdot 2^2 = 4+2+1+1/2+1/4 = 7.75$, and this is the nearest number to 11.1, also in $\mathcal{F}(2,5,-1,2)$.

Solution to Exercise 45. The smallest normalized number is always $2^{e_{min}}$. In case of single precision, this is 2^{-126}, for double precision, it is 2^{-1022}. Recall that the largest normalized number is

$$\left(1 - \left(\frac{1}{\beta}\right)^p\right) \beta^{e_{max}+1}.$$

For single precision, this yields

$$\left(1 - \left(\frac{1}{2}\right)^{24}\right) 2^{128} = 2^{128} - 2^{104}.$$

For double precision, we get

$$\left(1 - \left(\frac{1}{2}\right)^{53}\right) 2^{1024} = 2^{1024} - 2^{971}.$$

Solution to Exercise 46. For each exponent, $\mathcal{F}^*(\beta, p, e_{min}, e_{max})$ has $\beta - 1$ possibilities for the first digit, and β possibilities for the remaining $p-1$ digits. The size of $\mathcal{F}^*(\beta, p, e_{min}, e_{max})$ is therefore

$$2(e_{max} - e_{min} + 1)(\beta - 1)\beta^{p-1},$$

if we take the two possible signs into account.

$\mathcal{F}(\beta, p, e_{\min}, e_{\max})$ has extra nonnegative numbers of the form

$$0.d_1 \ldots d_{p-1} 2^{e_{\min}},$$

and there are β^{p-1} of them. Adding the non-positive ones and subtracting 1 for counting 0 twice, we get

$$2\beta^{p-1} - 1$$

extra numbers.

Solution to Exercise 47. The binary expansion of 0.1 is $0.0\overline{0011}$, obtained from the representation of 1.1 by subtracting 1. This value has to be rounded to the nearest value with 24 significant digits. Let us write out the expansion so that we get the first 26 significant digits of $0.0\overline{0011}$:

$$0.00011001100110011001100110011.$$

It follows that we have to round up to 1 at digit 24 to get the nearest `float` value

$$1.10011001100110011001101 \cdot 2^{-4}.$$

To see how this value differs from 0.1, let's convert it back into decimal representation. Interestingly, this is always possible without any error, since 0.1 (binary) is 0.5 (decimal), 0.01 (binary) is 0.25 (decimal), and so on. The decimal value that we obtain is

$$0.100000001490116119384765625.$$

Solution to Exercise 48. We compare floating point numbers for equality in `i != 1.0`, although one of them (namely the value of `i`) is the result of inexact computations, assuming a base-2 floating point number system. The inexactness comes from the rounding of 0.1 to a floating point number, and from the subsequent addition of numbers. In practice, this leads to an infinite loop, since `i != 1.0` will always be satisfied.

Solution to Exercise 49. We are adding very large to very small numbers during later steps of this loop. At some point, the value of `i` might have become so large that the increment by `1` has no effect anymore. We therefore get an infinite loop also in this case.

Solution to Exercise 50.

```
1  // Prog: dec2float.C
2  // compute the float representation of a number
3  // in the open interval (0,2)
4
5  #include <iostream>
6
```

```
 7  int main ()
 8  {
 9    // input
10    std::cout << "Decimal number x (0 < x < 2) =? ";
11    float x;
12    std::cin >> x;
13
14    // x = w * 2^e
15    float w = x;
16    int e = 0;
17
18    // as long as w < 1, decrement e and double w
19    for ( ; w < 1.0f; w *= 2.0f) --e;
20
21    // Now we have 1 <= w < 2, apply rule from lecture
22    std::cout << "Significand: ";
23    for ( ; w != 0.0; w = 2.0f * (w - int(w)))
24      std::cout << int(w);
25
26    std::cout << "\nExponent: " << e << "\n";
27
28    return 0;
29  }
```

Solution to Exercise 51. Here is the program based on the first formula.

```
 1  // Prog: pi1.C
 2  // approximate pi according to first n terms of the formula
 3  // pi = 4 - 4/3 + 4/5 - 4/7 ...
 4
 5  #include <iostream>
 6
 7  int main ()
 8  {
 9    // input
10    std::cout << "Number of iterations =? ";
11    unsigned int n;
12    std::cin >> n;
13
14    // computation (forward sum)
15    double pif = 0.0;
16    for (int i = 1; i < 2*n; i += 2)
17      if (i % 4 == 1)
18        pif += 4.0 / i;
19      else
20        pif -= 4.0 / i;
21
22    // computation (backward sum)
23    double pib = 0.0;
24    for (int i = 2*n-1; i > 0; i -= 2)
25      if (i % 4 == 1)
26        pib += 4.0 / i;
27      else
28        pib -= 4.0 / i;
29
30    // output
31     std::cout << "Pi is approximately "
32               << pif << " (forward sum), or "
33               << pib << " (backward sum); the difference is "
34               << pif - pib << "\n";
35
36    return 0;
37  }
```

When you run it for $n = 10,000$, for example, it gives on our platform the approximation 3.14139 (still off in the fourth digit after the decimal point). For $n = 100,000$, we get 3.14157 (still off in the fifth digit after the decimal point). For $n = 1,000,000$, finally, the result is correct to five digits after the decimal point: 3.14159.

Here is the approximation based on the second formula.

```
1  // Prog: pi2.C
2  // approximate pi according to the first n terms of the formula
3  // pi = 2 + 2*1 / 3 + 2*1*2 / 3*5 + 2*1*2*3 / 3*5*7
4
5  #include <iostream>
6
7  int main ()
8  {
9    // input
10   std::cout << "Number of iterations =? ";
11   unsigned int n;
12   std::cin >> n;
13
14   // auxiliary variables
15   // initialized for first term of forward sum (i=0)
16   double numer = 2.0; // numerator i-th term
17   double denom = 1.0; // denominator i-th term
18
19   // forward sum
20   // pif: value after term i (i=0 initially, then i=1,2,...,n-1)
21   double pif = 2.0;
22   for (int i = 1; i < n; ++i)
23     pif += (numer *= i) / (denom *= (2*i + 1)); // update to term i
24   // now numer and denom are the ones for i=n-1
25
26   // backward sum
27   // pib: value after term i (i=n-1 initially, then i=n-2,...,1,0)
28   double pib = numer / denom;
29   for (int i = n-1; i >= 1; --i) {
30     pib += (numer /= i) / (denom /= (2*i + 1)); // update to term i-1
31   }
32
33   // output
34   std::cout << "Pi is approximately "
35             << pif << " (forward sum), or "
36             << pib << " (backward sum); the difference is "
37             << pif - pib << "\n";
38
39   return 0;
40 }
```

This already gives the result 3.14159 for $n = 17$ on our platform, so this version is obviously preferable.

Solution to Exercise 52.

```
1  // Program: fpsys.C
2  // Provide a graphical representation of floating point numbers
3
4  #include <iostream>
5  #include <IFM/window>
```

```
 6
 7  int main()
 8  {
 9    // Input parameters of floating point system
10    std::cout << "Draw F*(2,p,e_min,e_max).\np =? ";
11    unsigned int p;
12    std::cin >> p;
13    std::cout << "e_min =? ";
14    int emin;
15    std::cin >> emin;
16    std::cout << "e_max =? ";
17    int emax;
18    std::cin >> emax;
19
20    // We compute significands using integral arithmetic, that is,
21    // scaled by 2^(p-1).
22
23    // compute the smallest normalized significand 2^(p-1)
24    unsigned int smin = 1;
25    for (unsigned int i = 1; i < p; ++i) smin *= 2;
26    // compute the largest normalized significand (2^p)-1
27    unsigned int smax = 2 * smin - 1;
28    // compute 2^emin
29    double pemin = 1;
30    for (int i = 0; i < emin; ++i) pemin *= 2;
31    for (int i = 0; i > emin; --i) pemin /= 2;
32    // compute 2^emax
33    double pemax = 1;
34    for (int i = 0; i < emax; ++i) pemax *= 2;
35    for (int i = 0; i > emax; --i) pemax /= 2;
36
37    // For each positive number x of the system draw a circle
38    // with radius x around the window center
39
40    // parameters to scale output
41    int cx = (ifm::wio.xmax() - ifm::wio.xmin()) / 2;
42    int cy = (ifm::wio.ymax() - ifm::wio.ymin()) / 2;
43    double scale = cx / (pemax * smax);
44
45    // zero
46    ifm::wio << ifm::Point(cx, cy);
47    // loop over all normalized significands
48    for (unsigned int i = smin; i <= smax; ++i)
49      // loop over all exponents
50      for (double m = pemin; m <= pemax; m *= 2)
51        ifm::wio << ifm::Circle(cx, cy, int(m * i * scale));
52
53    ifm::wio.wait_for_mouse_click();
54    return 0;
55  }
```

Solution to Exercise 53.

```
 1  // Prog: mandelbrot.C
 2  // draws (a part of) the Mandelbrot set and allows the user to
 3  // zoom in by clicking with the mouse on the region to be enlarged
 4  //
 5  // The Mandelbrot set is defined as the set of all complex numbers
 6  // c such that the complex iteration formula z := z^2 + c (starting
 7  // with z=0) always yields values z of absolute value at most two.
 8  // In the computations below, we perform a large but fixed number
 9  // of steps of this iteration for a given c; if all computed values
10  // are at most two in absolute value, we consider c as part of the
```

```
11  // Mandelbrot set (and depict its corresponding pixel in black),
12  // otherwise we draw a white pixel.
13
14  #include<IFM/window>
15
16  int main()
17  {
18    // the currently considered subset of the complex plane, initially
19    // [-2, 1] x [-1, 1] (covers the so-called main cardioid of the
20    // Mandelbrot set)
21    double r_min = -2; double r_max = 1;
22    double i_min = -1; double i_max= 1;
23
24    // window scaling factor; change this for larger/smaller display
25    // window
26    double window_scale = 500;
27
28    // zoom factor from one iteration to the next
29    double zoom_factor = 10;
30
31    // the display window dimensions in pixels (window should be
32    // congruent to the current complex plane subset)
33    int x_size = int (window_scale * (r_max - r_min));
34    int y_size = int (window_scale * (i_max - i_min));
35
36    // open the display window
37    ifm::Wstream w (x_size, y_size,
38                    "The Mandelbrot set (click to zoom in)");
39
40    // maximum number of iterations (the higher, the more accurate;
41    // the lower, the faster)
42    unsigned int max_iter = 500;
43
44    // main drawing loop; one iteration for every zoom scale
45    for(;;) {
46      // go through all pixels
47      for (int x=0; x<x_size; ++x)
48        for (int y=0; y<y_size; ++y) {
49
50          // compute corresponding point in complex plane
51          double r = r_min + x / window_scale;
52          double i = i_min + y / window_scale;
53
54          // do the Mandelbrot iteration for that point
55          // interpreted as complex number c = (r,i)
56          double r_z = 0; // z (real part)
57          double i_z = 0; // z (imaginary part)
58          unsigned int iter = 0;
59          while (iter < max_iter && r_z * r_z + i_z * i_z <= 4) {
60            // |z| <= 2; replace z by z^2 + c
61            double h = r_z * r_z - i_z * i_z + r; // new z_r
62            i_z = 2 * r_z * i_z + i;              // new z_i
63            r_z = h;
64            ++iter;
65          }
66          // coloring: max_iter -> black, other -> white
67          if (iter == max_iter)
68            w.set_color (w.number_of_colors()-2); // black
69          else
70            w.set_color (w.number_of_colors()-1); // white
71          w << ifm::Point (x, y);
72        }
73
74      // zoom in; new center is mouse click position
75      int x_c; int y_c;
```

```
76      w.get_mouse_click (x_c, y_c);
77      double r_c = r_min + x_c / window_scale;
78      double i_c = i_min + y_c / window_scale;
79      double r_span = r_max - r_min;
80      double i_span = i_max - i_min;
81      r_min = r_c - 0.5 * r_span / zoom_factor;
82      r_max = r_c + 0.5 * r_span / zoom_factor;
83      i_min = i_c - 0.5 * i_span / zoom_factor;
84      i_max = i_c + 0.5 * i_span / zoom_factor;
85      window_scale *= zoom_factor;
86      w.clear();
87    }
88
89    return 0;
90  }
```

Solution to Exercise 54. CGAL is the *Computational Geometry Algorithms Library*, an open source C++ library of data structures and algorithms for solving geometric problems. The CGAL homepage is `www.cgal.org`.

`CGAL::orientation` is a function that determines for three given points $p, q, r \in \mathbb{R}^2$ whether r lies to the left, on, or to the right of the oriented line through p and q. The resulting values (`CGAL::LEFTTURN`, `CGAL::COLLINEAR`, or `CGAL::RIGHTTURN`) define the orientation of the point triple $\{p, q, r\}$. `CGAL::LEFTTURN` means that p, q, r appear in counterclockwise order around the triangle spanned by p, q, r, while `CGAL::RIGHTTURN` signals clockwise order. `CGAL::COLLINEAR` means that all three points are on a common line, so the triangle is "flat".

The writer of the email is surprised since the orientation of a point triple should not change when all point coordinates are multiplied with a fixed scalar (in this case 100). But in reality, it does change, at least according to the function `CGAL::orientation`.

The reason is that the integer coordinates of the points $(14, 22), (15, 21), (19, 17)$ can be converted to `float` or `double` (we don't exactly know which of the two the writer of the email is using) without any error. In contrast, some of the coordinates of $(0.14, 0.22), (0.15, 0.21), (0.19, 0.17)$ don't have finite binary representations, so in converting them to `float` or `double`, errors are inevitable. Since the points are mathematically collinear (on the same line), it is clear that the tiniest error is enough to destroy this property. That's why `CGAL::orientation` delivers a result different from `CGAL::COLLINEAR`.

Here is what you could answer the writer of the email.

```
Hi,

assuming that you use type float or double to represent the point
coordinates, the inconsistency that you reported is due to the
conversion of point coordinates from the decimal input format to the
internally used binary format. Decimal integers like 14, 22 etc. can be
represented exactly in binary format, and CGAL::orientation returns
the correct answer CGAL::COLLINEAR for the three points with integer
```

coordinates. But decimal fractions like 0.14, 0.22 etc. do not necessarily have finite representations in binary format. This is like trying to write the number 1/3 as a decimal fraction. The best you can do is 0.33333... but wherever you stop, you make a small error.

Now, CGAL::orientation sees the points (0.14, 0.22), (0.15, 0.21) and (0.19,0.17) only after the conversion to binary format, and this conversion introduces some (tiny) errors. But since the points are mathematically collinear, even the tiniest errors may have the effect of destroying collinearity. This is exactly what you observed.

The problem is inevitable in working with floating-point numbers, since you cannot circumvent the decimal-to-binary conversion. All you can do is to only use point coordinates (integers, for example) for which the conversion is exact.