NAG Algorithms

The exactness and the speed of the numerical calculations done by computer-algebra systems are highly important. The same is true for any other sub field of mathematics. Chapter 8 covers three mathematical fields, in which the numerical calculations are dominant, that is, the solution of the nonlinear and linear systems of equation, the reliability of the calculations of the chaotic systems and the statistical evaluations. We are going to start this worksheet with the illustration of the numerical instability of the linear systems of equation. We will see that Maple 10 does not only execute the numerical calculations via software – as we could assume it based on the previous experiences – but it can also use hardware arithmetic, due to the built-in NAG routines.

For the exact calculations of the solutions of the linear systems of equation, consider the following simple, ill-conditioned problem.

$$x + 2\,y = 3\,,\ \ 1.00000001\,x + 2\,y = 3.00000001 \quad \Leftrightarrow \quad \begin{bmatrix} 1 & 2 \\ 1.00000001 & 2 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 3 \\ 3.00000001 \end{bmatrix}$$

We have written the system of equation in the form of equations and a matrix-vector. We created the second equation from the first in a way that the variable 1 of the x variable was increased by 10(-8) and we did the same with the constant located on the right side of the equation. It can be easily seen that the system of equation has only one solution:

$$x = 1 \text{ és } y = 1$$

The first equation is true according **to** the $1 + 2.1 = 3$ equation
and the second is true because of $1.00000001 \cdot 1 + 2 \cdot 1 = 3.00000001$

.

Let's see what the LinearSolve procedure of the LinearAlgebra package does with the system of equation. For this, give the M matrix of the system with the Matrix procedure. The b constant vector on the right side can be created with the Vector procedure.

```
> restart;
> M := Matrix(1..2, 1..2, [[1, 2], [1.00000001, 2]]);
```

$$M := \begin{bmatrix} 1 & 2 \\ 1.00000001 & 2 \end{bmatrix} \quad (1)$$

```
> b := Vector([3, 3.00000001]);
```

$$b := \begin{bmatrix} 3 \\ 3.00000001 \end{bmatrix} \quad (2)$$

The LinearSolve quickly solves the system of equation. In order to get more information about the steps and the procedures called during the solution of the linear system of equation, the infolevel variable of

the LinearAlgebra package should be set at 1. In this case, Maple shows us the methods used and informs us about the procedures called.

```
> infolevel[LinearAlgebra] := 1;
```

$$infolevel_{LinearAlgebra} := 1 \quad (3)$$

```
> < x, y >= LinearAlgebra[LinearSolve](M, b);
LinearSolve:   "using method"   LU
LinearSolve:   "calling external function"
LinearSolve:   "NAG"   hw_f07adf
LinearSolve:   "NAG"   hw_f07aef
```

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 1.00000002220446071 \\ 0.99999998889776954 \end{bmatrix} \quad (4)$$

We have approximately received the $\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$ solution we expected. The scale of the difference is approximately 10-8 which coincides with the scale of the perturbation given for the constants in the second equation. Now let's interpret the text information returned.

The LinearSolve informed us that it had used the LU method during the solution of the system of equation. This means that it uses the division of the M=L.U product for the solution. In this case L is a lower and U is an upper triangle-shaped matrix. We can read more about this on the definition help site of the LU division.

```
> help(Definition, LUdecomposition);
```

However, let's continue detailing the responses of the LinearSolve procedure. The "calling external function" informed us about what was going to happen and we received the names of functions: "hw_f07adf" and "hw_f07aef". The NAG before the names of the functions is the abbreviation of the Numerical Algorithm Group. So if Maple users purchase the system then the numerical algorithms of the NAG are automatically at their disposal.

Let's see the names of the procedures. The names of both procedures start with the characters "hw", which means that these procedures use hardware arithmetic for their calculations. We have already known that the calculations of Maple are exact due to the software arithmetic. And in this case we can see that it uses hardware arithmetical routines. How does the system choose between the two types of arithmetic?

In case the double-point, hardware representation of numbers of the computer is enough for the exactness of the calculations, then it counts this way because the calculations are much faster. But if we need a better exactness than the double exactness of the hardware, the system uses the software representation of numbers with its unlimited exactness. In chapter 1.4 we illustrated how the software exactness of the calculations could be affected by the joint usage of the evalf procedure and the Digits environment variable.

Can we affect during the calculations which arithmetic Maple should use and when? Naturally, yes, we can. We can regulate with the UseHardwareFloats environment variable if Maple should use the hardware or the software arithmetic for the calculations. After entering the UseHardwareFloats:=true instruction, Maple executes the numerical operations with the hardware arithmetic of our computer. Obviously, the UseHardwareFloats:=false uses the software arithmetic.

Query the present settings of the UseHardwareFloats environment variable.

```
> 'UseHardwareFloats' = UseHardwareFloats;
```

$$UseHardwareFloats = deduced \quad (5)$$

So there is a third state of the UseHardwareFloats environment variable and this is the “deduced”. In this case Maple decides which arithmetic it uses and when, based on the value of the Digits variable and the exactness of the double word hardware arithmetic available on the computer. If the value of the Digits is smaller than or equal to the hardware exactness available then Maple calculates with the hardware arithmetic because it is quicker and more exact. However, if we want to calculate with exactness bigger than the properties of the hardware, that is, the value of the Digits is set bigger than the hardware exactness, then the system chooses the software arithmetic.

A question can arouse at this point: how big is the hardware exactness of our computer and how can we gain information about it? The exactness of the hardware arithmetic can be queried by the evalhf (Digits) instruction. In the case of a 32 byte computer architecture, its evaluated value is 15 significant digits according to the creators of Maple:

```
> Gépünk hardware pontossága = evalhf(Digits);
```

$$Gépünk\ hardware\ pontossága = 14. \quad (6)$$

The evalhf (“evaluate using hardware floating-point”) procedure is used to execute the evaluations with the help of the hardware arithmetic of the computer, irrespectively of the settings of the UseHardwareFloats environment variable.

Compare the evaluations of the $\sqrt{2}$ *returned* **by** *the evalf* **and** *evalhf procedures.*

```
> sqrt(2) = evalf(sqrt(2));
> sqrt(2) = evalhf(sqrt(2));
```

$$\sqrt{2} = 1.414213562$$

$$\sqrt{2} = 1.41421356237309515 \quad (7)$$

While the evalf procedure gave 10, the evalhf returned 18 significant digits. Since the default value of the Digits environment variable is 10, the evalf shows only 10 significant digits of the result and rounds at the last value. To understand the response of the evalhf, let’s see the call of the evalf but exactly for 19 digits.

```
> sqrt(2) = evalf(sqrt(2), 19);
```

$$\sqrt{2} = 1.414213562373095049 \quad (8)$$

We can see that the calculation of the evalhf procedure is exact only until 16 significant digits. Naturally, it has returned more significant digits but we cannot trust those subsequent to the 15th digit because of the exactness limit of the hardware arithmetic.

Can we calculate all the values known so far by using the hardware tools? For example, let’s calculate the values of the GAMMA and BesselJ functions mentioned in chapter 4.3 with both methods.

```
> ('GAMMA')(5) = evalf(GAMMA(5));
```

$$\Gamma(5) = 24. \tag{9}$$

```
> ('GAMMA')(5) = evalhf(GAMMA(5));
```

$$\Gamma(5) = 24.0000000000000036 \tag{10}$$

```
> ('BesselJ')(5, 1) = evalf(BesselJ(5, 1), 10);
```

$$\mathrm{BesselJ}(5, 1) = 0.0002497577302 \tag{11}$$

```
> evalhf(BesselJ(5, 1));
Error, remember tables are not supported in evalhf
```

Well, it has its limits. The evalf worked in both cases but the evalhf informed us via an error message that the remember table of the procedure does not support this function. According to this, the values of the Bessel functions cannot be calculated with hardware arithmetic. If there are functions the values of which the system cannot calculate with software arithmetic then which are the functions supported by the evalhf? Let's ask the system itself.

```
> help(evalhf, fcnlist);      # the output of the instruction is not shown
```

The response is that the functions supported by the C language can be calculated with the evalhf because Maple uses the built-in functions of the C language similar to the hardware for the calculations. Since the GAMMA function was implemented in the C language but the BesselJ function was not, we received a value in the first case and an error message in the second case.

Let's return to the initial, ill-conditioned 2x2 linear system of equation. Solve it with software arithmetic. For this, set the UseHardwareFloats environmental variable to false then solve the M.x=b linear system of equation again.

```
> UseHardwareFloats := false;
```

$$\mathit{UseHardwareFloats} := \mathit{false} \tag{12}$$

```
> < x, y >= LinearAlgebra_LinearSolve(M, b);
LinearSolve:   "using method"   LU
LinearSolve:   "calling external function"
LinearSolve:   "NAG"   sw_f07adf
LinearSolve:   "NAG"   sw_f07aef
```

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 1.000000000 \\ 1.000000000 \end{bmatrix} \tag{13}$$

Comparing with the results of the run of (4), the difference is that the "hw" letters have been changed to "sw", the letters of software. We have also received the x=1, y=1 exact solution. It seems that the creators of the NAG routines wrote both the hardware and software versions of all the procedures. Since the results of (4) and (13) are different but possibly the algorithms are not that's why the difference derives from the difference of the two representations of numbers. This is confirmed by a research in which we compare the product of the M matrix and the $\begin{bmatrix} 1 \\ 1 \end{bmatrix}$ solution vector first with the routines using the software, then with the hardware exactness.

```
> M.Vector([1, 1]) = b;
MatrixVectorMultiply:   "calling external function"
MatrixVectorMultiply:   "NAG"   sw_f06paf
```

$$\begin{bmatrix} 3. \\ 3.00000001 \end{bmatrix} = \begin{bmatrix} 3 \\ 3.00000001 \end{bmatrix} \quad (14)$$

```
> UseHardwareFloats := deduced;
```

$$UseHardwareFloats := deduced \quad (15)$$

```
> M.Vector([1, 1]) = b;
MatrixVectorMultiply:   "calling external function"
MatrixVectorMultiply:   "NAG"   hw_f06paf
```

$$\begin{bmatrix} 3. \\ 3.00000000999999994 \end{bmatrix} = \begin{bmatrix} 3 \\ 3.00000001 \end{bmatrix} \quad (16)$$

The result of the first multiplication is the b vector because Maple used the software multiplication. (See the "sw" initial letters of the name of the procedure of the "sw_f06paf" matrix-vector multiplier.) Then we set the UseHardwareFloats environment variable to the initial "deduced" value. Thus Maple used the hardware arithmetic in the second case because the Digits=10 value is smaller than the hardware arithmetic with the 14 decimal exactness. In the second case the b vector was not returned but its approximation within hardware exactness.

The linear algebra calls this type of M matrix ill-conditioned which is described by the condition number. If the condition number is big then the matrix is ill-conditioned. The number of a matrix condition can be calculated by the ConditionNumber procedure of the LinearAlgebra package. Furthermore, we know that an nxn linear system of equation can be solved and its solution is determined if the determinant of the matrix of the system is zero. (Cramer's rule). Let's see what the determinant is in this case.

```
> kondició_szám = LinearAlgebra[ConditionNumber](M);
ConditionNumber:   "calling external function"
ConditionNumber:   "NAG"   hw_f06raf
ConditionNumber:   "NAG"   hw_f07adf
ConditionNumber:   "NAG"   hw_f07agf
```

$$kondició\_szám = 6.000000064\ 10^{8} \quad (17)$$

```
> Determináns = LinearAlgebra[Determinant](M)
```

$$Determináns = -2.\ 10^{-8} \quad (18)$$

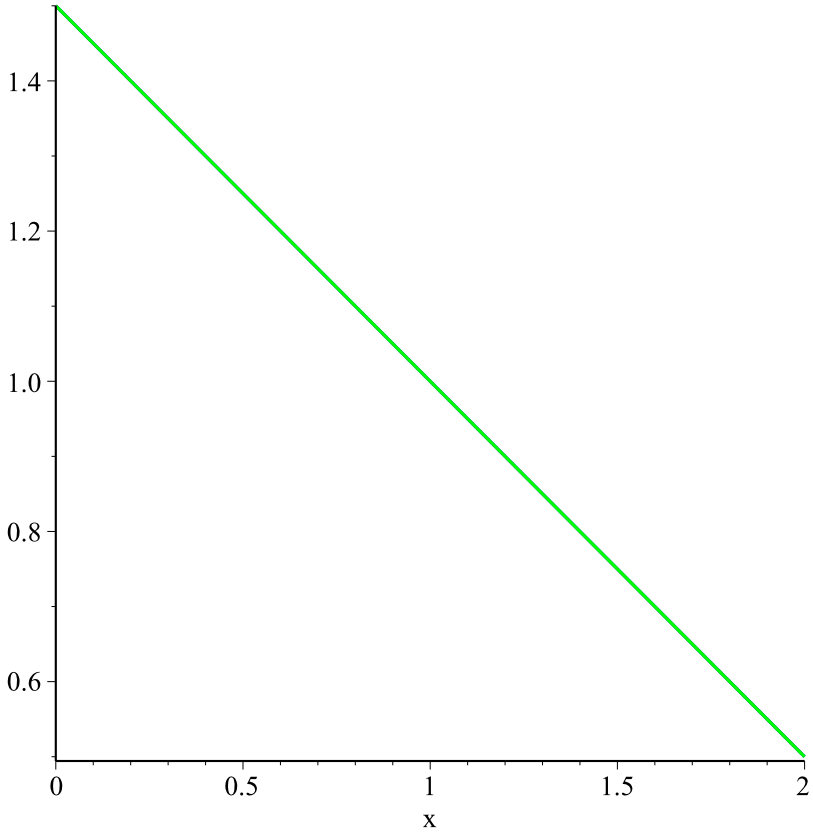
The condition number of the M matrix is 6.108, which competes with the Digits=10 setting, but lags behind the 14 significant digits exactness of the hardware arithmetic. With this simple, ill-conditioned example we were able to reveal the problem. The value of the determinant is so small that it approximates the software zero. Finally, for the sake of illustration plot the 3D lines determined by the equations near the root locations x=1 and y=1.

```
> convert(M.Vector([x, y]) - b, list); egyenesek := [rhs(isolate(%1, y)), rhs(isolate(%2, y))];
```

```
> plot(egyenesek, x = 0 ..2, color = [blue, green]);
VectorScalarMultiply:   "calling external function"
VectorScalarMultiply:   "NAG"   hw_f06edf
```

$$[x + 2\,y - 3.,\ 1.00000001\,x + 2\,y - 3.00000000999999994]$$

$$egyenesek := \left[-\frac{1}{2}\,x + 1.500000000,\ -0.5000000050\,x + 1.500000005\right]$$



Well, we can only see one line instead of two. The reason for this is that the two lines in the graph completely coincide with each other. Although their one and only intersection point is the x=1, y=1 point, they proceed so close to each other that they cannot be distinguished from each other. Thus it is difficult to exactly isolate the intersection point and we can also make a serious mistake at the numerical solution of the system of equation.

That's all about the exactness and trustworthiness of the numerical calculations. And now let's see the speed of the calculations.

The speed of the numerical calculations should be tested on a considerable amount of data. For this, we are using the two images below. The first image shows a bulb working while the other shows its explosion. Our aim is to fit them to the same size, put them onto each other with 50% transparency and display them. During the conversions we will measure the length of the calculations and the amount of the memory space used to prove the efficiency of the numerical calculations of Maple.

For the solution of the task, the ImageTools package is essential to use because it contains the most important procedures concerning the handling of the digital images. Thus we start the work with the loading of the package.

```
> restart;
> with(ImageTools) :
```

The first image is the “bulb1.jpg” file. We put its exact directory path and its file name into the “imagefile” string variable while paying attention to keep the conventions of the operation system. After this, we have the file read into the “image1” variable with the Read procedure of the ImageTools package.

```
> kepfile1 := "c:/klima/fejezet6_10/izzo1.jpg"
```

$$kepfile1 := \text{"c:/klima/fejezet6\_10/izzo1.jpg"} \quad \textbf{(19)}$$

```
> kep1 := Read(kepfile1)
```

$$kep1 := \begin{bmatrix} 1..300\ x\ 1..300\ x\ 1..3\ Array \\ Data\ Type:\ float_8 \\ Storage:\ rectangular \\ Order:\ C\_order \end{bmatrix} \quad \textbf{(20)}$$

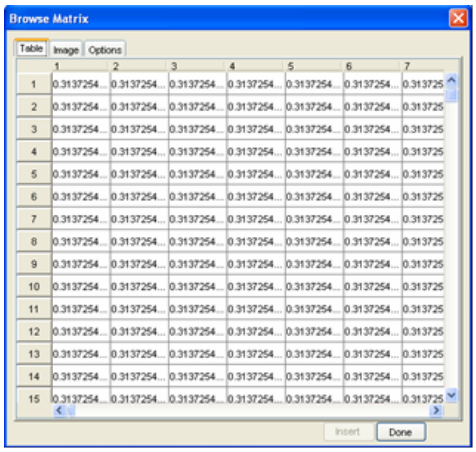
After the execution of the Read procedure, the “image1” 3D array has been created. Its size is 300x300x3 so the array contains 270000 elements. All of its elements are float8 floating point numbers that can be represented in 8 bytes. The elements are stored in a rectangle shape arrangement in an order coinciding with the C language. The placeholder, displayed as output, contains this information. So far it seems good but how can we look at the other elements of the array?

If we double click on the field showing the properties of the array then the window on the right side disappears. We can see the elements of the array arranged in a rectangle shape. Since the array consists of 3 layers put on each other, we can see the other layers as well by clicking the Options tab.	

The Options window shows that according to the default, the first index was put into the rows and the second index was put into the columns. Thus we can see a small part of a chart containing 300 rows and columns. The window shows the layers of the first element because the third dimension is set to 1 in the Options window. In case we set this value to 2 and switch to the Table tab we can see the elements of the second layer. We can get the data of the third layer the same way.

We can change the layout of the rows and columns with the Options tab. For instance, if we leave the first index in the rows but we put the third index into the columns, then we can see a chart containing 300 rows and 3 columns. In this case we have to check 300 charts put on each other if we want to see all the data.

We can display this huge chart as an image with the Image tab. We can also use different colouring techniques. Slowly the image of the bulb starts to appear.



We can make the data of the image1 array visible with the printf procedure. The following instruction displays the triple values between the 91st and 100th column of the 100th row.

```
> printf("%10.4f\n", kep1[100, 91..100])
    0.2078     0.1647     0.1490
    0.1804     0.1529     0.1294
    0.3922     0.3647     0.3412
    0.8392     0.8275     0.8000
    1.0000     1.0000     0.9725
    0.9804     0.9843     0.9608
    0.9922     0.9922     0.9922
    0.9882     0.9922     1.0000
    0.9882     0.9961     0.9922
    0.9882     0.9961     0.9922
```

The FormatFromName procedure of the ImageTools package gives the type of the image file given as a parameter. The dimension limits previously determined can be queried with the rtable_dims procedure.

```
> FormatFromName(kepfile1)
```

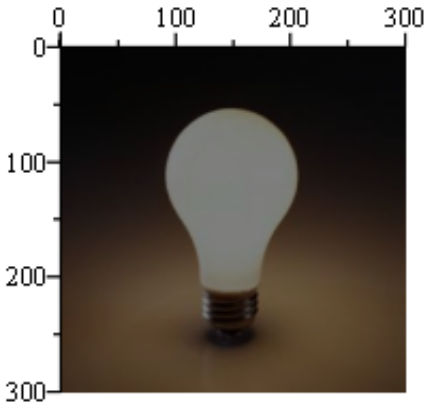
JPEG **(21)**

```
> dim1 := [rtable_dims(kep1)]
```

dim1 := [1..300, 1..300, 1..3] **(22)**

We know that the extension of the image is JPG and its size is 300x300x3. Let's display the image with the Preview procedure of the ImageTools package.

```
> Preview(kep1)
```



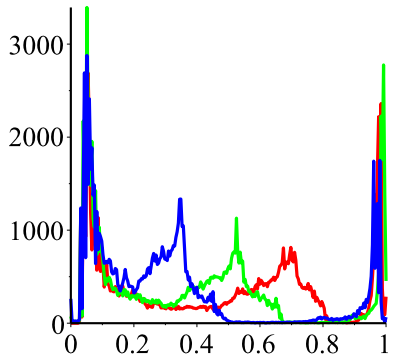
We can see that the image consists of 300 pixels both in the x and y directions. So the first and second dimension of the image1 array coincide with the x and y coordinates of the pixels in a way that the upper left corner is the [0,0] point. The third dimension of the array gives the colour. According to this, every colour can be given by three independent numbers between 0 and 1. These three colour channels describe the intensity of Red, Green and Blue. The frequency of the basic colours of the image can be displayed with the PlotHistogram procedure.

Since the basic colours can be given with numbers between 0 and 1, as we experienced at the display of the elements of the array, thus the [0,1] interval appears on the horizontal axis of the histogram.

Maple divides the [0,1] interval to considerable amount of sub intervals and counts how many parts of each layer of the array are put into certain small sub intervals. Every layer contains 300x300=90000 numbers. We can see that in the case of red, green and blue, the frequency values close to 0 and 1 are big. The extreme maximum of the frequency of the values close to 0 are around 3300, while the maximum of the frequency of the values close to 1 reaches 2770.

When the component value of each colour is 0, that is, the [0,0,0] value is displayed then the colour black is returned. The [1,1,1] colour composition creates the colour white. So the dark

```
> PlotHistogram(kep1)
```



colours are dominant in the image but the very light spots are also frequent. We have known this but now we can see them expressed in numbers.	

Let's repeat the instructions executed so far with the "bulb2.jpg" but at every step

- measure the CPU usage time necessary for the execution of the instruction
- measure the memory space used during the run and
- calculate the speed of the execution.

The CPU time is measured by putting the value of the time() function into the variable named beginning. It gives the CPU time elapsed since the start of the worksheet. When we finish, we extract the current time() value from it. This difference gives the CPU time necessary for the execution of the instruction measured in seconds.

The size of the memory used by Maple is given by the MemoryInUse procedure of the MmaTranslator [Mma] package. Neither this nor the time procedure has a parameter. The memory used during the execution of the instruction can be received if we extract the value before the start of the instruction from the value of the MemoryInUse() after the instruction. The memory space should be measured in kbyte. By the Units(SI) palette, we can paste units. By choosing the [[s]] we insert the unit of the second. Since the byte is not on the list thus we insert the [[unit]] symbol then change the unit text to byte. It does not consider the kbyte an official unit. That's why we write the letter k before the sign of the unit which is the shortened form of kilo.

```
> kepfile2 := "c:/klima/fejezet6_10/izzo2.jpg"
```

$$kepfile2 := \text{"c:/klima/fejezet6\_10/izzo2.jpg"} \quad (23)$$

```
> with(MmaTranslator[Mma])
```

$$[AbsoluteTime, BitAnd, BitNot, BitOr, BitXor, Chop, ClearAttributes, CompoundExpression, Decompose, Dimensions, DirectoryName, Distribute, Drop, FactorInteger, Fit, FixedPoint, FixedPointList, FromDate, FromDigits, Get, Insert, LeafCount, Level, MapAll, MemoryInUse, PadLeft, PadRight, Partition, PolynomialReduce, Precision, ReadList, RealDigits, ReplaceRepeated, Sequence, Show, Split, StringForm, StringPosition, ToDate, Unique, Which, WriteString] \quad (24)$$

Let's see the time of the calculation, the memory need and the speed of the operation for the display of the image.

```
> kezdet := time( ) : memoria1 := MemoryInUse( ):
  kep2 := Read(kepfile2);
```

```
> tartam, memoria := (time( ) - kezdet)[[s]], evalf( (MemoryInUse( ) - memoria1)/1024 )k[[byte]];
```

$$kep2 := \begin{bmatrix} 1..270\ x\ 1..250\ x\ 1..3\ Array \\ Data\ Type: float_8 \\ Storage: rectangular \\ Order: C\_order \end{bmatrix}$$

$$tartam, memoria := 1.131\ [\![s]\!], 1641.796875\ k[\![byte]\!] \quad (25)$$

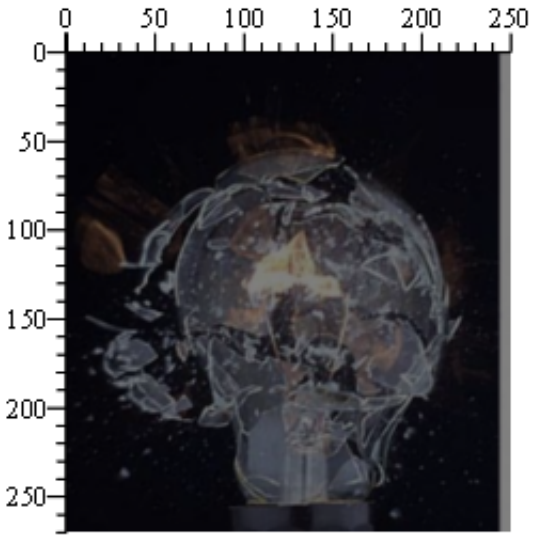
Read has read the image of the second bulb into a 270x250x3=202500 array. All the elements of the Array are 8 bit=1 byte thus the full size of the image is 202500 byte = 197.75 kbyte. The Read procedure has reserved the 8th fold of the minimum memory needed for the array. The speed is more than 1000 kbyte/sec.

> $sebesség = \frac{memoria}{tartam}$

$$sebesség = \frac{1451.632958\ k[\![byte]\!]}{[\![s]\!]} \quad (26)$$

Let's see the time of the calculation, the memory need and the speed of the operation for the display of the image.

> $kezdet := time() : memoria1 := MemoryInUse() :$
> $Preview(kep2);$
> $tartam,\ memoria := (time() - kezdet)[\![s]\!],\ evalf\left(\frac{MemoryInUse() - memoria1}{1024}\right) k[\![byte]\!];$
> $sebesseg = \frac{memoria}{tartam}$



$$tartam,\ memoria := 2.353\ [\![s]\!],\ 2031.285156\ k\,[\![byte]\!]$$

$$sebesseg = \frac{863.2746094\ k\,[\![byte]\!]}{[\![s]\!]} \qquad (27)$$

We can see that the x and y size of the second image is smaller than of the first. Let's reduce the x and y size of the first image in a way that both pictures should be the same size. Calculate the scale of the reductions.

> $dim2 := [rtable\_dims(kep2)]$

$$dim2 := [1..270,\ 1..250,\ 1..3] \qquad (28)$$

> $xr := op(2, dim2[1])/op(2, dim1[1]);$
> $yr := op(2, dim2[2])/op(2, dim1[2]);$

$$xr := \frac{9}{10}$$

$$yr := \frac{5}{6} \qquad (29)$$

The Scale procedure of the ImageTools package does the scaling. Only the name of the image needs to be given. Now we have to reserve more memory and load the image to a smaller space with a difficult image resizing algorithm.

> $kezdet := time(\,) : memoria1 := MemoryInUse(\,) :$
$atmeretezett := Scale\left(kep1, xr, yr\right);$
$tartam,\ memoria := (time(\,) - kezdet)\,[\![s]\!],\ evalf\left(\frac{MemoryInUse(\,) - memoria1}{1024}\right) k[\![byte]\!];$
$sebesseg = \frac{memoria}{tartam}$

$$atmeretezett := \begin{bmatrix} 1..270\ x\ 1..250\ x\ 1..3\ Array \\ Data\ Type:\ float_8 \\ Storage:\ rectangular \\ Order:\ C\_order \end{bmatrix}$$

$$tartam,\ memoria := 1.602\,[\![s]\!],\ 1579.429688\ k[\![byte]\!]$$

$$sebesseg = \frac{985.9111660\ k[\![byte]\!]}{[\![s]\!]} \qquad \textbf{(30)}$$

Since the size of the first and second images is the same, create the arithmetic mean of certain array elements. Maple has to do 202500 additions and multiplications by the 0.5. It is definitely not an easy task.

> $kezdet := time(\,) :\ memoria1 := MemoryInUse(\,) :$
$egyuttes := 0.5 \cdot (atmeretezett + kep2);$
$tartam,\ memoria := (time(\,) - kezdet)\,[\![s]\!],\ evalf\left(\frac{MemoryInUse(\,) - memoria1}{1024}\right) k[\![byte]\!];$
$sebesseg = \frac{memoria}{tartam}$

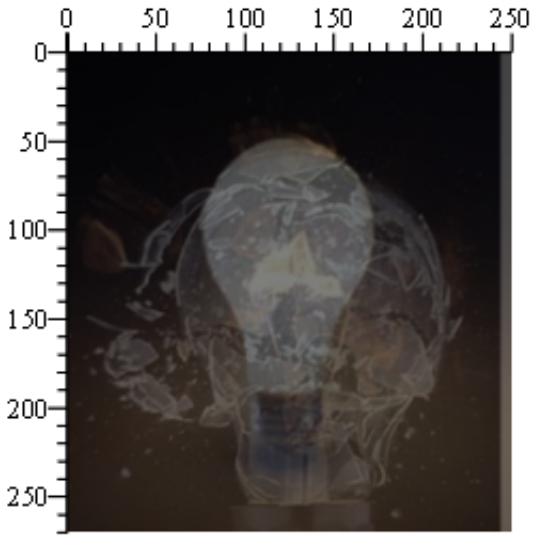
$$egyuttes := \begin{bmatrix} 1..270\ x\ 1..250\ x\ 1..3\ Array \\ Data\ Type:\ float_8 \\ Storage:\ rectangular \\ Order:\ C\_order \end{bmatrix}$$

$$tartam,\ memoria := 1.252\,[\![s]\!],\ 1576.480469\ k[\![byte]\!]$$

$$sebesseg = \frac{1259.169704\ k[\![byte]\!]}{[\![s]\!]} \qquad \textbf{(31)}$$

With this procedure, we have put the two images onto each other with 50-50% transparency. The Preview procedure confirms this.

> $kezdet := time(\,) :\ memoria1 := MemoryInUse(\,) :$
$Preview(egyuttes);$
$tartam,\ memoria := (time(\,) - kezdet)\,[\![s]\!],\ evalf\left(\frac{MemoryInUse(\,) - memoria1}{1024}\right) k[\![byte]\!];$
$sebesseg = \frac{memoria}{tartam}$



$$tartam, memoria := 1.432\ [\![s]\!],\ 2026.714844\ k\ [\![byte]\!]$$

$$sebesseg = \frac{1415.303662\ k\ [\![byte]\!]}{[\![s]\!]} \qquad (32)$$

Naturally, after each run we get a different time and speed value. However, based on the results concerning the time needed for the operations and their speed, we can conclude that quick algorithms operate in the background.

We have mentioned the mechanism of the creation of the RGB colour composition but we have not displayed an image based on it. As an example, let's display a 30x30 red-white-green flag.

First, we give a generator function with which we can easily create the elements of the array to be created. The upper 10x30 area of the flag will be red so 1 has to be entered into these spots of the first layer of the array and 0 has to be entered into the other two. The middle part of the flag is white which can be created by setting the value of the three layers of this field of the array to 1. The lower 10x30 field of the flag is red so 1 has to be entered into the second layer of this field of the array and 0 has to be written to the other two.

```
> restart; with(ImageTools) : with(MmaTranslator[Mma]):
```

```
> szinek:=(i,j,k)->if (i<=10) and (k=1) then 1
>                  elif (i>20) and (k=2) then 1
>                  elif (i>10) and (i<=20) then 1 else 0 end if:
```

> $kezdet := time() : memoria1 := MemoryInUse() :$

> $zaszlo := Array(1..30, 1..30, 1..3, szinek, datatype = float_8, order = C\_order);$

> $tartam,\ memoria := (time() - kezdet)[\![s]\!], \quad evalf\left(\frac{MemoryInUse() - memoria1}{1024}\right) k[\![byte]\!] ;$

$$sebesség = \frac{memoria}{tartam}$$

$$zaszlo := \begin{bmatrix} 1..30\ x\ 1..30\ x\ 1..3\ Array \\ Data\ Type: float_8 \\ Storage: rectangular \\ Order: C\_order \end{bmatrix}$$

$$tartam, memoria := 1.302\ [\![s]\!], 99.93359375\ k[\![byte]\!]$$

$$sebesség = \frac{76.75391225\ k[\![byte]\!]}{[\![s]\!]} \qquad (33)$$

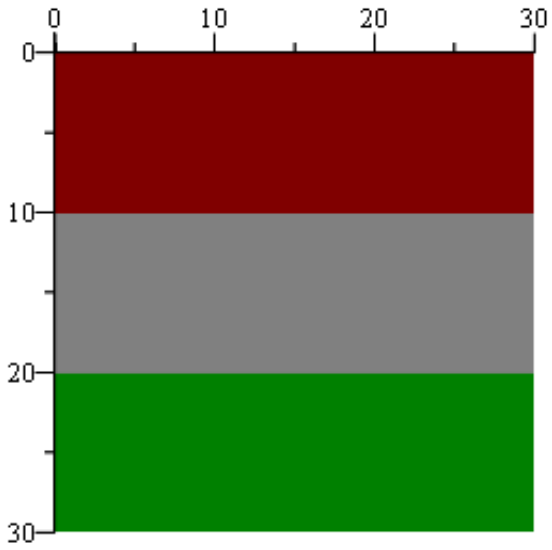
The creation of the array called flag calls for explanation. The first three parameters give the dimensions and the fourth parameter is a function which fills the array created with values. The datatype=float8 and order=C_order options should also be given because these are not defaults and the procedures handling the images require these kinds of data structures.

> $kezdet := time() : memoria1 := MemoryInUse() :$

> $Preview(zaszlo);$

> $tartam,\ memoria := (time() - kezdet)[\![s]\!], evalf\left(\frac{MemoryInUse() - memoria1}{1024}\right) k[\![byte]\!] ;$

$$sebesség = \frac{memoria}{tartam}$$



$$tartam, memoria := 1.681\ [\![s]\!],\ 2199.089844\ k\ [\![byte]\!]$$

$$sebesség = \frac{1308.203358\ k\ [\![byte]\!]}{[\![s]\!]} \qquad (34)$$

We have got to know how to measure the exactness and speed of the calculations and algorithms. We do hope that we have been able to improve your computer-algebra knowledge and arouse your interest in the mathematical problem solving with computer-algebra systems.

As the sources are infinite, we recommend you three help sites where you can find useful information about the numerical calculations.

```
> help(LinearAlgebra, General, EffNumLA);
> help(Statistics, Computation);
> help(Optimization, General, Computation);
```