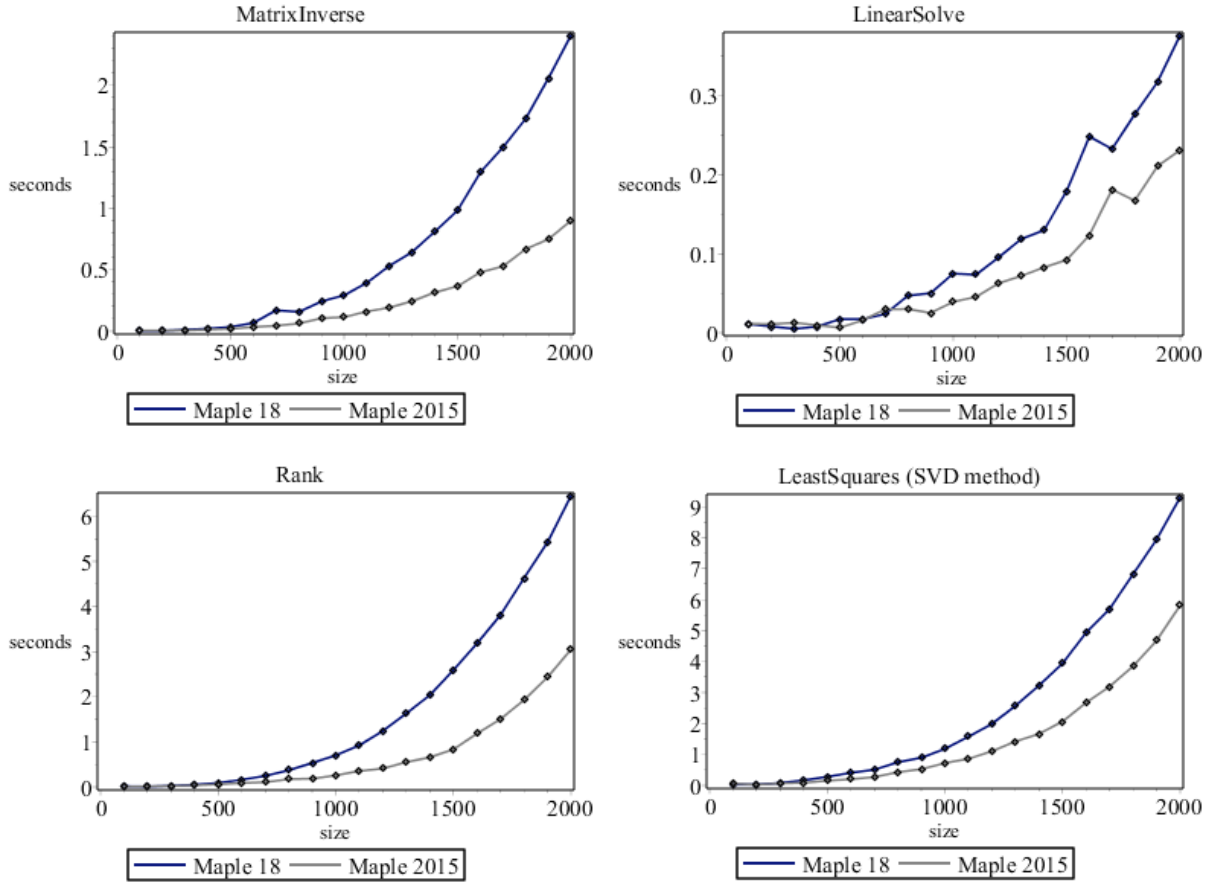
Performance

Floating-point LinearAlgebra

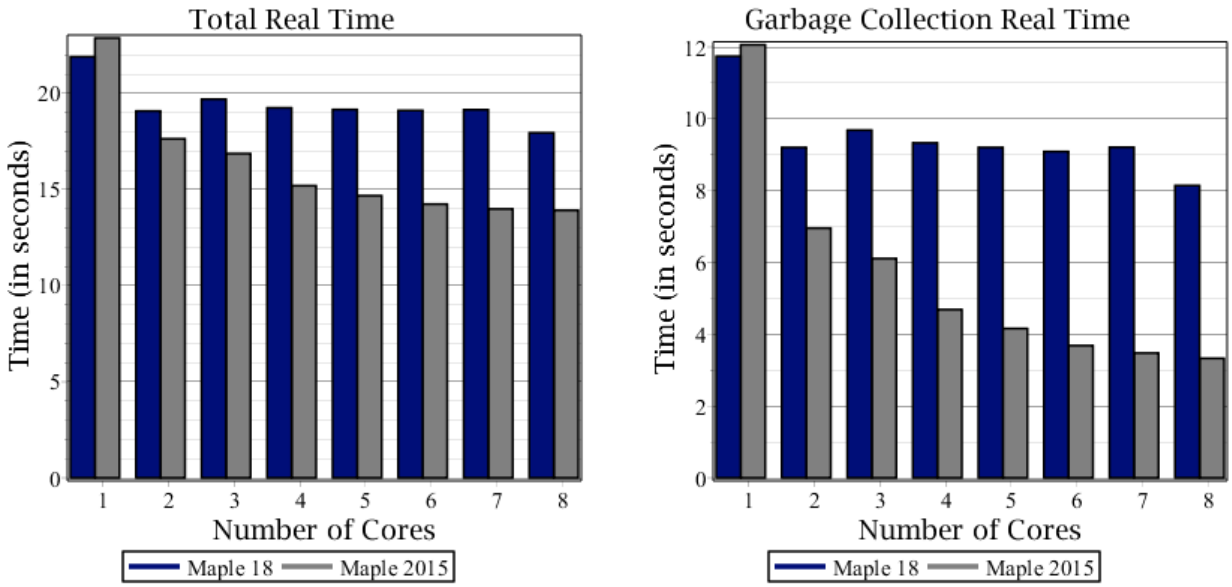
The performance of several commands in the LinearAlgebra package for floating-point operations done at default double precision is improved in Maple 2015 on the 64-bit Windows platform. The improvement is due to updating the version and the usage of the Intel Math Kernel Library (MKL).

The following plots were computed in Windows 7 64-bit on a machine with a 3.50 GHz AMD FX-8320 8-core processor. The degree of improvement of some commands may vary with the machine hardware.



Improved Parallel Garbage Collector

Maple garbage collector has been improved to perform more operations in parallel. This allows for greater parallelism during collection, leading to better performance. The following graphs compare the performance between Maple 18 and Maple 2015.



The graph in the left compares performance of the total time for the benchmark (as mentioned later) vs. the number of cores used. The graph in the right compares performance of the garbage collector vs. the number of cores. The improved collector in Maple 2015 shows better parallel performance, especially as additional cores are added, whereas Maple 18 tends to plateau quite quickly.

This benchmark can be re-run using the code from the following Code Edit Region. The graphs shown earlier were generated using 10 iterations and Array size of 10^6.

```
Generate benchmark data
```

Multivariate Modular Gcds

The performance for computing greatest common divisors of multivariate polynomials modulo of a machine size prime number is increased by orders of magnitude. The improvement is due to a new optimized C-level implementation based on evaluation and interpolation. On 64-bit platforms, the following examples are more than 20 times faster in Maple 2015 than in Maple 18 on the same machine.

> $p := prevprime(2^{62})$

$$p := 4611686018427387847 \quad \textbf{(3.1)}$$

> $f_2 := Expand((x+y+2)^{70}) \bmod p$:

> $g_2 := Expand(f_2^2) \bmod p$:

> $h_2 := Expand(f_2 \cdot (f_2+1)) \bmod p$:

> $degree(g_2), nops(g_2)$

$$140, 10011 \quad \textbf{(3.2)}$$

> $degree(h_2), nops(h_2)$

$$140, 10011 \quad \textbf{(3.3)}$$

> $F_2 := CodeTools:\text{-}Usage(Gcd(g_2, h_2) \bmod p)$:
$evalb(F_2 = f_2)$

```
memory used=109.80KiB, alloc change=0 bytes, cpu time=16.00ms,
real time=20.00ms, gc time=0ns
```

$$true \quad \textbf{(3.4)}$$

> $f_3 := Expand((x+y+z+2)^{15}) \bmod p$:

> $g_3 := Expand(f_3^2) \bmod p$:

> $h_3 := Expand(f_3 \cdot (f_3+1)) \bmod p$:

> $degree(g_3), nops(g_3)$

$$30, 5456 \quad \textbf{(3.5)}$$

> $degree(h_3), nops(h_3)$

$$30, 5456 \quad \textbf{(3.6)}$$

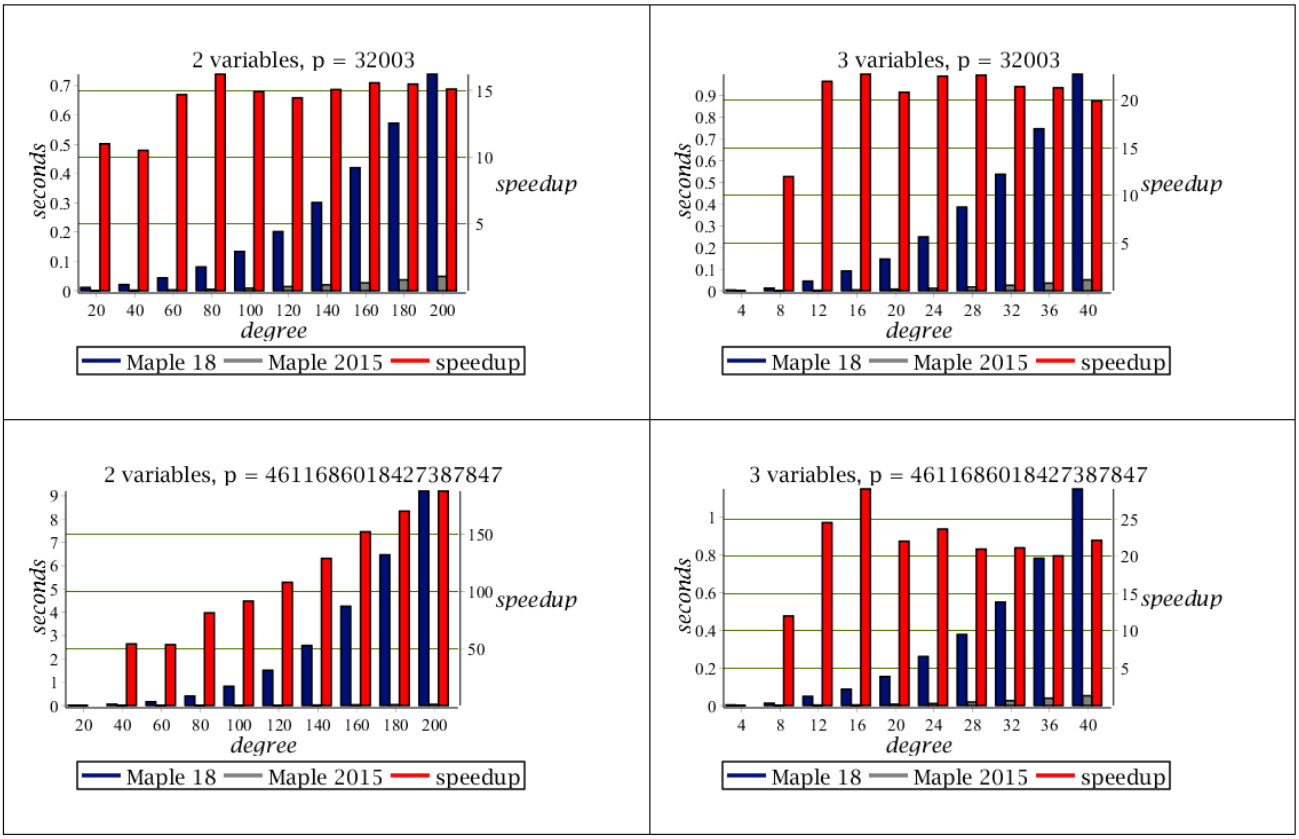
> $F_3 := CodeTools:\text{-}Usage(Gcd(g_3, h_3) \bmod p)$:
$evalb(F_3 = f_3)$

```
memory used=16.70KiB, alloc change=0 bytes, cpu time=31.00ms,
real time=30.00ms, gc time=0ns
```

$$true \quad \textbf{(3.7)}$$

Code for generating plots

Improvements are significant for smaller primes as well. Following are some benchmarks, taken on an AMD Opteron 62xx class CPU running at 2.1GHz. In all cases, the input polynomials are dense of degree d and the gcd has degree $d/2$.



More special functions implemented in *evalhf*

The Airy functions, AiryAi and AiryBi, and the LambertW function are now implemented in evalhf, for fast hardware precision evaluation. For the Airy functions, this implementation is restricted to real arguments (evalhf will call back to the standard library for complex arguments with non-0 imaginary part). For the LambertW function, all branches are implemented in evalhf for real and complex arguments.

```
> evalhf(AiryAi(3.5))
```

$$0.0025840987869896374 \tag{4.1}$$

```
> evalhf(AiryBi(1,-2.7))
```

$$-0.42989534308201538 \tag{4.2}$$

```
> evalhf(LambertW(3, 25.1234))
```

$$0.37284323213604109 + 17.300307507976740\,I \tag{4.3}$$

More consistent evaluation of elementary and special functions in *evalf*, *evalhf*, and hardware floats

Much effort has been put into making the results of evaluating expressions involving floating point numbers consistent across the three evaluation modes: software float evaluation with evalf, hardware float evaluation with evalhf, and hardware float evaluation of expression involving HFloat objects (Maple objects which hold hardware floating point numbers). The correct and consistent evaluation of such expressions where branch cuts are involved is of particular concern. For example, consider the following:

```
> arcsin(1.5), arcsin(1.5 + 0. I), arcsin(1.5 - 0. I)
```

$$1.570796327 - 0.9624236501\,I,\ 1.570796327 + 0.9624236501\,I,\ 1.570796327 - 0.9624236501\,I \tag{5.1}$$

```
> evalhf(arcsin(1.5)), evalhf(arcsin(1.5 + 0. I)), evalhf(arcsin(1.5 - 0. I))
```

$$1.57079632679489656 - 0.962423650119206942\,I,\ 1.57079632679489656 + 0.962423650119206942\,I,\ 1.57079632679489656 - 0.962423650119206942\,I \tag{5.2}$$

```
> arcsin(HFloat(1.5)), arcsin(HFloat(1.5 + 0. I)), arcsin(HFloat(1.5 - 0. I))
```

$$1.57079632679490 - 0.962423650119207\,I,\ 1.57079632679490 + 0.962423650119207\,I,\ 1.57079632679490 - 0.962423650119207\,I \tag{5.3}$$

In previous versions of Maple, the second and third sets of results were all the same, as the hardware float computation environments of Maple were not respecting the branch cut.