Language and Programming

Workbook (.maple) Files

In addition to working with the new Workbook (.maple) files in the Standard Interface, **.maple** files can be used in the same way as **.mla** files, for example with savelib and the setting of libname.

Modular Arithmetic

Several improvements were made to the modp1 function, which is a low-level data structure and library implementing algorithms for univariate polynomials modulo a prime.

The use of bit level programming techniques results in an overall speedup of about 40% on small problems.

```
> p := modp1(Prime(1));
```

$$p := 3037000453$$

```
> f := modp1(ConvertIn(randpoly(x,degree=20,dense),x),p):
> g := modp1(ConvertIn(randpoly(x,degree=20,dense),x),p):
> TIMER := time():
  for i to 100000 do h := modp1(Multiply(f,g),p): end do:
  time()-TIMER;
```

$$0.416$$

For larger polynomials modulo smaller primes, Kronecker substitution is used. This example runs about 3.3 times faster.

```
> p := 32003;
```

$$p := 32003$$

```
> f := modp1(ConvertIn(randpoly(x,degree=5000,dense),x),p):
> g := modp1(ConvertIn(randpoly(x,degree=5000,dense),x),p):
```

```
> TIMER := time():
  for i to 100 do h := modp1(Multiply(f,g),p): end do:
  time()-TIMER;
```

```
                              0.708
```

An asymptotically fast division algorithm was added. This example runs about 5 times faster.

```
> p := 32003;
```

$$p := 32003$$

```
> f := modp1(ConvertIn(randpoly(x,degree=2000,dense),x),p):
```

```
> g := modp1(ConvertIn(randpoly(x,degree=1000,dense),x),p):
```

```
> TIMER := time():
  for i to 100 do h := modp1(Rem(f,g,'q'),p); end do:
  time()-TIMER;
```

```
                              0.479
```

Fast algorithms were added for multiplication and division with multiprecision primes. These examples run about 20 and 9 times faster, respectively.

```
> p := 2^150-3;
```

$$p := 1427247692705959881058285969449495136382746621$$

```
> f := modp1(ConvertIn(randpoly(x,degree=1000,dense,coeffs=rand(0.
  .p-1)),x),p):
```

```
> g := modp1(ConvertIn(randpoly(x,degree=1000,dense,coeffs=rand(0.
  .p-1)),x),p):
```

```
> TIMER := time():
  for i to 100 do h := modp1(Multiply(f,g),p); end do:
  time()-TIMER;
```

```
> TIMER := time():
  for i to 100 do r := modp1(Rem(h,f,'q'),p); end do:
  time()-TIMER;
```

```
                              4.419
                              9.101
```

You can see the impact of these improvements in Maple's top level commands as well.

This factorization modulo p runs about twice as fast.

```
> n := 200:
> p := nextprime(ceil(2^n*Pi));
```

$$p := 5048344754617993871973410141242436836214643421488662971535423$$

```
> f := x^n + x + 1:
> F := CodeTools:-Usage( Factor(f) mod p ):
memory used=248.02MiB, alloc change=8.00MiB, cpu time=3.34s, real
time=3.35s, gc time=154.80ms
```

MultiSet

Maple 2016 has a new data structure called the MultiSet. The basic structure can be used to manage unordered collections of data while accounting for multiple occurrences of particular members. For example, a collector of baseball cards might have 3 "Ty Cobb" cards, 1 "George Brett", and 2 "Nolan Ryan". This collection can be represented as a MultiSet:

> $MyCards := MultiSet(\ [\text{"Ty Cobb"}, 3], [\text{"George Brett"}, 1], [\text{"Nolan Ryan"}, 2]\ )$

$$MyCards := \{[\text{"Ty Cobb"}, 3], [\text{"George Brett"}, 1], [\text{"Nolan Ryan"}, 2]\}$$

If the collector then obtains 2 more "Ty Cobb" cards and a "Reggie Jackson", these cards can be added to the collection with the following command:

> $MyCards := MyCards + \{[\text{"Ty Cobb"}, 2], [\text{"Reggie Jackson"}, 1]\}$

$$MyCards := \{[\text{"Ty Cobb"}, 5], [\text{"George Brett"}, 1], [\text{"Nolan Ryan"}, 2], [\text{"Reggie Jackson"}, 1]\}$$

For a standard MultiSet, the multiplicity of any element must be a non-negative integer (elements with multiplicity 0 are removed from the MultiSet). A more general structure, allowing any real number to represent the "multiplicity" is available by attaching the index generalized to the MultiSet call:

> $M := MultiSet[generalized]\left(a = \frac{1}{2}, b = -3.1415, c = 2 \right)$

$$M := \left\{\left[a, \frac{1}{2}\right], [b, -3.1415], [c, 2]\right\}$$

This kind of MultiSet can be produced by the convert command when applied to a rational function:

> $convert\left(\frac{(x^2+1)^3 (x-2)^2}{(x^2+3x-1)^4}, MultiSet, generalized \right)$

$$\{[x-2, 2], [x^2+1, 3], [x^2+3x-1, -4]\}$$

indets[flat]

The indets function has a new option, flat, which stops recursion into sub-expressions once a matching subexpression has been found.

In the following, the regular indets command returns the indeterminates of both sin(x) and its children, while using the flat option stops sooner.

$indets(\sin(x))$

$$\{x, \sin(x)\}$$

$indets[flat](\sin(x))$

$$\{\sin(x)\}$$

The next pair of commands contrasts the default behavior with the flat option when given a type to look for. The sub-expression *f(4,5)* is omitted from the flat result because it is contained inside another expression that is already part of the result.

$indets([f(1, 2) + f(3, f(4, 5))], function)$

$$\{f(1,2), f(3, f(4,5)), f(4,5)\}$$

$indets[flat]([f(1, 2) + f(3, f(4, 5))], function)$

$$\{f(1,2), f(3, f(4,5))\}$$

The following commands repeat the above observation. The *[x]* result does not occur in the flat version because the whole input expression is a list.

$indets([f(g([x]))], list)$

$$\{[x], [f(g([x]))]\}$$

$indets[flat]([f(g([x]))], list)$

$$\{[f(g([x]))]\}$$

userinfo for Grid:-Map and Grid:-Seq

By setting infolevel you can now see the partition size and message information when executing parallel Map and Seq commands.

$infolevel[Grid{:-}Map] := 4;$

$$4$$

$data := LinearAlgebra{:-}RandomVector(1000) :$

$myfunc := \mathbf{proc}(x)\ \sin(x \cdot \mathrm{LambertW}(x))\ \mathbf{end};$

$$\mathbf{proc}(x)\ \sin(x * \mathrm{LambertW}(x))\ \mathbf{end\ proc}$$

$Grid{:-}Set(myfunc);$

$Grid{:-}Map(myfunc, data) :$

```
Map: using tasksize=13; number of elements=1000; number of
partitions=77
Map: sending node 0 elements 1 .. 13
```

```
Map: sending node 1 elements 14 .. 26
Map: sending node 2 elements 27 .. 39
Map: sending node 3 elements 40 .. 52
Map: sending node 0 elements 53 .. 65
Map: sending node 1 elements 66 .. 78
Map: sending node 2 elements 79 .. 91
Map: sending node 0 elements 92 .. 104
Map: sending node 1 elements 105 .. 117
Map: sending node 2 elements 118 .. 130
Map: sending node 0 elements 131 .. 143
Map: sending node 1 elements 144 .. 156
Map: sending node 2 elements 157 .. 169
Map: sending node 0 elements 170 .. 182
Map: sending node 1 elements 183 .. 195
Map: sending node 2 elements 196 .. 208
Map: sending node 3 elements 209 .. 221
Map: sending node 0 elements 222 .. 234
Map: sending node 1 elements 235 .. 247
Map: sending node 2 elements 248 .. 260
Map: sending node 3 elements 261 .. 273
Map: sending node 0 elements 274 .. 286
Map: sending node 1 elements 287 .. 299
Map: sending node 0 elements 300 .. 312
Map: sending node 2 elements 313 .. 325
Map: sending node 1 elements 326 .. 338
Map: sending node 0 elements 339 .. 351
Map: sending node 2 elements 352 .. 364
Map: sending node 1 elements 365 .. 377
Map: sending node 0 elements 378 .. 390
Map: sending node 2 elements 391 .. 403
Map: sending node 1 elements 404 .. 416
Map: sending node 0 elements 417 .. 429
Map: sending node 2 elements 430 .. 442
Map: sending node 1 elements 443 .. 455
Map: sending node 0 elements 456 .. 468
Map: sending node 1 elements 469 .. 481
Map: sending node 2 elements 482 .. 494
Map: sending node 0 elements 495 .. 507
Map: sending node 1 elements 508 .. 520
Map: sending node 0 elements 521 .. 533
Map: sending node 2 elements 534 .. 546
Map: sending node 1 elements 547 .. 559
Map: sending node 0 elements 560 .. 572
Map: sending node 2 elements 573 .. 585
Map: sending node 1 elements 586 .. 598
Map: sending node 0 elements 599 .. 611
Map: sending node 2 elements 612 .. 624
Map: sending node 1 elements 625 .. 637
Map: sending node 0 elements 638 .. 650
Map: sending node 2 elements 651 .. 663
Map: sending node 0 elements 664 .. 676
Map: sending node 1 elements 677 .. 689
Map: sending node 2 elements 690 .. 702
Map: sending node 0 elements 703 .. 715
Map: sending node 1 elements 716 .. 728
Map: sending node 0 elements 729 .. 741
Map: sending node 2 elements 742 .. 754
```

```
Map: sending node 1 elements 755 .. 767
Map: sending node 0 elements 768 .. 780
Map: sending node 2 elements 781 .. 793
Map: sending node 1 elements 794 .. 806
Map: sending node 0 elements 807 .. 819
Map: sending node 2 elements 820 .. 832
Map: sending node 0 elements 833 .. 845
Map: sending node 1 elements 846 .. 858
Map: sending node 0 elements 859 .. 871
Map: sending node 2 elements 872 .. 884
Map: sending node 0 elements 885 .. 897
Map: sending node 1 elements 898 .. 910
Map: sending node 0 elements 911 .. 923
Map: sending node 2 elements 924 .. 936
Map: sending node 0 elements 937 .. 949
Map: sending node 1 elements 950 .. 962
Map: sending node 2 elements 963 .. 975
Map: sending node 0 elements 976 .. 988
Map: sending node 1 elements 989 .. 1000
Map: done sending all partitions
Map: received results from all nodes; collating result
```

$Grid{:}\text{-}Map[tasksize = 250](myfunc, data)$:

```
Map: using tasksize=250; number of elements=1000; number of
partitions=4
Map: sending node 0 elements 1 .. 250
Map: sending node 1 elements 251 .. 500
Map: sending node 2 elements 501 .. 750
Map: sending node 3 elements 751 .. 1000
Map: done sending all partitions
Map: received results from all nodes; collating result
```

Overloading Element-wise Operations

In Maple, an operator followed by a tilde (~) represents an element-wise application of that operator. For example, matrix multiplication is implied by the dot operator, but .~ or *~ mean element-wise multiplication.

$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \cdot \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$

$$\begin{bmatrix} 7 & 10 \\ 15 & 22 \end{bmatrix}$$

$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} .\sim \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$

$$\begin{bmatrix} 1 & 4 \\ 9 & 16 \end{bmatrix}$$

New in Maple 2016 is the ability to overload element-wise operations when using objects. This can be done in one of two ways.

1. Overload the ~ operator:

In this method the object exports a procedure named `~`. All element-wise operations involving objects of this type will be dispatched through this procedure. Since the data is a container, the implementation simply unpacks the object data from each of the arguments and then calls the global element-wise operator, :-`~`.

```
> unprotect('Obj');
  module Obj() option object;
     export data, `~`;
     data := <1,2;3,4>;
     `~` := proc()
          local newargs, operation, i;
          operation := op(procname);
          newargs := seq(`if`(i::thismodule,i:-data,i),i=args);
          return :-`~`[operation](newargs);
     end;
  end module:
```

```
> Obj +~ <1,1;2,2>;
```

$$\begin{bmatrix} 2 & 3 \\ 5 & 6 \end{bmatrix}$$

```
> <10,10; 10,10>  *~ Obj;
```

$$\begin{bmatrix} 10 & 20 \\ 30 & 40 \end{bmatrix}$$

2. Overload the index operation `?[]`, and the `upperbound` method:

In this case the data held by the object is a list of vectors. It is not in a format that is able to be passed to the top-level tilde command. Rather than overload `~`, and because indexing is already defined for this object via `?[]`, adding `upperbound` allows Maple to query the size of the data so it can properly perform the element-wise operations for the object.

```
> unprotect('Obj2');
  module Obj2() option object;
     export `?[]`, upperbound, data;
     data := [<1,2>, <3,4>];
     `?[]` := proc(me,idx::[posint,posint])
          me:-data[idx[1]][idx[2]];
     end proc;
```

```
    upperbound := proc(me,dim:=NULL)
        if dim = NULL then
            op(map(:-upperbound,me:-data));
        else
            :-upperbound(me:-data[dim],1);
        end if;
    end proc;
  end module:
```

```
> Obj2[2,1];
```

$$3$$

```
> upperbound(Obj2);
```

$$2, 2$$

```
> Obj2 +~ <1,1;2,2>;
```

$$\begin{bmatrix} 2 & 3 \\ 5 & 6 \end{bmatrix}$$

```
> <10,10; 10,10>  *~ Obj2;
```

$$\begin{bmatrix} 10 & 20 \\ 30 & 40 \end{bmatrix}$$

Forgetting Remember Table Entries

Procedures can make use of option remember or option cache to store computed values. When called a second time with the same arguments, the "remembered" or "cached" value will be returned rather than computing it again.

```
> myfun := proc( x )
     option cache;
     printf("performing long computation...\n");
     return HAIRY(x);
  end proc:
```

```
> myfun(z^2);
performing long computation...
```

$$HAIRY(z^2)$$

```
> myfun(z^2);
```

$$HAIRY(z^2)$$

Notice that the second call to myfun did not print a message. The return value was fetched from the cache rather than recomputed.

The forget command has been given new functionality to selectively clear items from all procedure caches and remember tables. It works when given .. (dot dot) as the first argument (an ellipsis representing all procedures). The second argument should be a variable, denoting which items to selectively clear. All procedures containing a reference to that variable will be cleared.

```
> forget(..,z);

> myfun(z^2);
performing long computation...
```

$$HAIRY(z^2)$$

As demonstrated, the cached result for the input z^2 was forgotten, and thus recomputed when myfun was called again. The input had a reference to z, and so did the output. Either is enough to cause the reference to be forgotten.

Normally clearing remember tables for all procedures in the system is not necessary. There are, however, some situations where this may be useful (for example, when working with global variables with attributed properties).

Word Lists

The EssayTools package, which has an assortment of commands for analyzing essays and words, now makes it easier to get access to the built-in words list. Two new commands, GetWordList and GetWordTable, give access to the word list in different forms. As before, AppendToWordList and SetWordList allow customizing your own word lists.

$words := EssayTools:\text{-}GetWordList(\ ) :$

$numelems(words);$

$$113898$$

$words[10200\,..10210];$

["tallyhoed", "copperplate", "wowing", "gads", "ago", "olfactory", "band's", "rye", "tibiae", "gulch", "creep"]

$wordtab := EssayTools:\text{-}GetWordTable(\ ) :$

if *wordtab*["wonderfull"] ≠ 1 **then** *EssayTools:-SpellCorrectWord*("wonderfull"); **end if**;

"wonderful"

Program Analysis

The CodeTools package provides a ProgramAnalysis subpackage which provides tools for analyzing programs to determine if they are suitable for parallelization and to help formally verify that a program meets its specification.

- New commands for analyzing the data dependencies in for-loops will determine when they can be parallelized, and transformations can be applied to resolve these dependency issues and enable the loop's parallelization.
- The invariants of a while-loop can be computed and, when combined with the pre-condition and guard condition of the loop, used to verify whether or not the post-condition will be satisfied.