Numerical computations in MuPAD use multiprecision software floats. Therefore the software arithmetic in MuPAD is much slower than the arithmetic in special purpose numerical tools that are based on compiled C or Fortran code supported by hardware floats. With MuPAD Release 2.5, the numerical Scilab package was linked to the MuPAD system. This allows to send numerical tasks to the external Scilab tool for fast processing using hardware floats.

What is Scilab?

Scilab is a numerical package developed at INRIA Rocquencourt (http://www-rocq.inria.fr/scilab). As a standalone tool, it provides an interactive user interface to communicate with the Scilab interpreter. Numerous functions for numerical analysis are available, some of which are written in the Scilab language. Others are mere interfaces to C and Fortran routines from standard numerical packages such as LINPACK or, in the current versions of Scilab, LAPACK.

Scilab provides basic functionality such as matrix data types and their manipulation as well as various special libraries and toolboxes. In these libraries, numerous routines exist for various applications areas such as control theory, linear algebra, optimization, statistics etc.

The Generic MuPAD–Scilab Link

Overview

MuPAD 2.5 is available with or without the Scilab package as an add-on. If Scilab is available, MuPAD establishes a link to the Scilab system by starting an (invisible) external Scilab process without a window and sending/receiving data to/from the Scilab kernel. Technically, this is realized by a dynamic module providing a domain scilab to the MuPAD user. A Scilab routine xyz, say, is available in MuPAD as a method scilab::xyz. It can be called like any other MuPAD function.

The link is generic: Apart from the Scilab graphics, all functions in the Scilab installation are available and can be called from MuPAD. The idea is that the input data for Scilab routines are created as MuPAD objects. Calling a function scilab::xyz(data) converts the MuPAD data to a suitable collection of floating point data which are sent to the external Scilab process together with the information which Scilab routine is to be called. Scilab processes the data with the requested Scilab routine and sends the result back to MuPAD. There, the data are received, converted to MuPAD objects and handed to the current MuPAD session in which the result can be further processed by MuPAD commands.
The documentation of the MuPAD–Scilab link (call \texttt{scilab} in the MuPAD session) provides a conversion table between MuPAD data types and corresponding Scilab data types. E.g., MuPAD matrices of type \texttt{DOM_ARRAY} or \texttt{Dom::Matrix()} are converted to (dense) Scilab matrices; MuPAD matrices of type \texttt{Dom::SparseMatrix()} are converted to sparse Scilab matrices.

The generic link between MuPAD and Scilab boosts MuPAD’s functionality dramatically: \textit{All} Scilab functions have become part of the MuPAD system and can by used like the functions of any other native MuPAD library.

The only drawback of this approach is that the user has to know the name and the calling syntax (number and types of input parameters) of the Scilab function that he wishes to use. \textit{Some acquaintance with Scilab is required!}

\textbf{Overhead}

There certainly is some overhead using MuPAD as an interface to Scilab:

1) Before sending a MuPAD object to Scilab, it has to be checked for suitability. All operands in the MuPAD object are converted to floats. It is checked that this was successful and that all floats are of a suitable size such that they can be represented in Scilab.
2) Some time is needed for sending and receiving the data to/from Scilab.
3) The result computed by Scilab has to be converted to a MuPAD object.

Whilst the communication overhead 2) is almost negligible, the time needed for the conversions 1) and 3) can be observed and measured. Experiments have shown that for certain problems such as operations on large $n \times n$ matrices, the overhead typically amounts to the same order of magnitude as the computational costs in Scilab. In particular, for algorithms with costs $O(n^3)$ such as matrix multiplication, inversion, eigenvalue computations etc., the computing costs in Scilab exceed the overhead of the conversions if the data are sufficiently large.

In any case, when comparing the timings needed by Scilab together with the overhead with the timings needed by MuPAD’s software floats, it is worthwhile to use the MuPAD–Scilab link even for small numerical problems. E.g., for the multiplication of $n \times n$ float matrices, the MuPAD–Scilab link beats MuPAD’s software floats for matrices as small as $n = 5$ (on LINUX systems).

\textbf{Demonstration}

Let us give some demonstration of the generic link. First, we let Scilab generate a random float matrix of dimension $200 \times 200$. The matrix can be processed by functions of the MuPAD libraries. We compute the trace of $A$ via \texttt{linalg::tr}:

\begin{verbatim}
MuPAD

>> A := scilab::rand(200, 200):
>> linalg::tr(A)

Output

98.62451434

\end{verbatim}

For computing the inverse of $A$, we may use MuPADs matrix arithmetic by calling $A^{-1}$. Note, however, that matrix inversion is a rather expensive algorithm ($O(n^3)$ for $n \times n$ matrices):
It pays off to let Scilab do the work. The Scilab function for inverting a matrix is `inv`:

```
>> rtime(scilab::inv(A))*msec
```

Output

```
753 msec
```

In Scilab, some functions need to assign their return values to a special “left hand side”. An example is the QR factorization of a matrix, which has to be called in the form `[Q, R] = qr(A)` in Scilab. Without an assignment to a list `[Q, R]` of the matrix factors, the Scilab function `qr` raises an error. The generic MuPAD–Scilab link assumes simple function calls without assignments to special “left hand sides” and thus calls `qr` in an illegal way:

```
>> scilab::qr(A)
```

Output

```
[FAIL, "Scilab Error", 41.0]
```

In such a case, a small Scilab program has to be sent. A Scilab program can be generated in MuPAD via the method `scilab::func`:

```
>> f := scilab::func(["A"], ["Q", "R"], ["[Q, R] = qr(A)"]):
```

In this declaration, the input parameter is the matrix `A`, the return values are the matrices `Q` and `R`. The program itself is a single Scilab command calling the `qr` function and assigning the result to the Scilab list `[Q, R]`. The MuPAD function `f` can now be called in the MuPAD session, returning the sequence of return values `Q` and `R` specified in `scilab::func`:

```
>> [Q, R] := [f(A)]:
```

We use Scilab to compute the determinants of `Q`, `R` and `A`:

```
>> scilab::det(Q), scilab::det(R), scilab::det(A)
```

Output

```
-1.0, -2.364152509e80, 2.364152509e80
```
With `scilab::func`, arbitrarily complex Scilab routines may be defined. In the following example, we consider the numerical integration of an ordinary differential equation. The initial value problem to be solved is:

\[ \frac{d}{dt} y(t) = f(t, y(t)) = t \cdot \sin(y(t)) + \cos(y(t)), \quad y(t_0) = y_0. \]

In Scilab, the numerical integrator is the routine `ode`. It requires the right hand side `f(t, y)` of the differential equation as a procedure. Thus, the Scilab code needs to consist of two steps: i) declaring `f(t, y)` as a scilab procedure, ii) passing this procedure together with the initial conditions to `ode`.

Correspondingly, the following code generated via `scilab::func` consists of two command lines:

i) The Scilab routine `deff` is used for the on-line definition of a suitable Scilab function; `deff` has to be called with Scilab strings (enclosed by single quotes).

ii) The numerical integrator `ode` is called.

```
MuPAD
>> Sci := scilab::func(["t0", "y0", "t"], ["y"],
>>    ["deff('ydot = f(t, y)', 'ydot = t*sin(y) + cos(y)');
>>      y = ode(y0, t0, t, f)
>>    "]):
```

In MuPAD, one may declare a corresponding integrator by the following steps:

```
MuPAD
>> f := (t, y) -> [t*sin(y[1]) + cos(y[1])]:
>> Mup := (t0, y0, t) -> numeric::odesolve(f, t0..t, [y0]):
```

We call both the Scilab integrator and MuPAD’s ODE solver `numeric::odesolve` and compare the results:

```
MuPAD
>> t0 := 0: y0 := 0:
>> for t in [1, 10, 100] do
>>    ts := rtime((ys := Sci(t0, y0, t)))*msec:
>>    tm := rtime((ym := Mup(t0, y0, t)))*msec:
>>    print(Unquoted, "t" = t, Scilab = ys, ts, MuPAD = ym, tm);
>> end_for:
```

Output
```
t = 1, Scilab = 1.134477454, 5 msec, MuPAD = [1.134477255], 63 msec
t = 10, Scilab = 3.040908148, 11 msec, MuPAD = [3.040908158], 719 msec
t = 100, Scilab = 3.131592018, 24 msec, MuPAD = [3.131591987], 14848 msec
```

**Summary:** The generic MuPAD–Scilab link allows to call arbitrary Scilab functions from a MuPAD session. Also arbitrary Scilab procedures can be created and launched from a MuPAD session. Despite the communication overhead between MuPAD and Scilab, the use of the link is worthwhile even for small numerical tasks.
Hardware Floats hidden in MuPAD’s numeric library

The generic MuPAD–Scilab link described before requires some acquaintance with Scilab: the user has to know the name and the calling syntax of the Scilab function appropriate for the task.

In order to make the use of fast numerics easier for devoted MuPAD users, Scilab was hidden in some routines of MuPAD’s numeric library in such a way that no knowledge of Scilab functions and their calling syntax is required.

Scilab uses double precision floats which corresponds to an arithmetic with about 16 decimal digits. If the MuPAD environment variable DIGITS determining the numerical working precision has a value smaller than 16, the following numeric routines call Scilab internally if it is available:

- numeric::det: determinant of a matrix
- numeric::eigenvalues: eigenvalues of a matrix
- numeric::eigenvectors: eigenvectors of a matrix
- numeric::expMatrix: exponential function for matrices
- numeric::factorCholesky: Cholesky factorization of a matrix
- numeric::factorLU: LU factorization of a matrix
- numeric::factorQR: QR factorization of a matrix
- numeric::fft: fast Fourier transform
- numeric::fMatrix: functional calculus for matrices
- numeric::invfft: inverse fast Fourier transform
- numeric::inverse: inverse of a matrix
- numeric::leastSquares: least squares solution of an overdetermined linear system
- numeric::linsolve: solution of a linear system given by equations
- numeric::matlinsolve: solution of a linear system given by a coefficient matrix
- numeric::singularvalues: singular values of a matrix
- numeric::singularvectors: singular value decomposition of a matrix

If the precision goal set by DIGITS is 16 or more decimal places, or if Scilab is not available, these functions use the software arithmetic of the MuPAD kernel to compute their result. The user notices the difference only by the time needed to get the answer.

Supported by hardware floats, computing the eigenvalues of a $300 \times 300$ matrix is a matter of seconds:

```plaintext
MuPAD

>> A := linalg::hilbert(300):
>> t := rtime():
>> eigenvals := numeric::eigenvalues(A);
>> (rtime() - t)*msec

Output

[2.322019937, 1.033217168, 0.3433362682, ..., -3.147852847e-16]

3314 msec
```

The use of hardware floats can be suppressed via the option SoftwareFloats. The hardware floats used internally in the previous call turn out to be about 100 times faster than MuPAD’s software floats:
With Scilab in the background, the routine `numeric::matlinsolve` is the fastest numerical linear solver available in MuPAD 2.5. Using MuPAD’s new domain for sparse matrices, there is no problem to generate and solve huge sparse systems $A \cdot x = b$. We consider the tridiagonal system

$$
\begin{pmatrix}
2 & -1 & 0 & & & & \\
-1 & 2 & -1 & & & & \\
 & -1 & \ddots & \ddots & & & \\
& & \ddots & -1 & & & \\
0 & & & 0 & 2 & -1 & \\
\end{pmatrix}
\begin{pmatrix}
x_1 \\
x_2 \\
\vdots \\
x_n \\
\end{pmatrix}
= 
\begin{pmatrix}
1 \\
1 \\
\vdots \\
1 \\
\end{pmatrix}
$$

which arises in the discretization of second order differential equations. We solve $10^4$ equations for $10^4$ unknowns:

Also MuPAD’s software floats can handle the task, needing more than 50 times the runtime of the previous call:

See the article *Dom::SparseMatrix – A domain for sparse matrices in MuPAD* by Kai Gehrs on page 69 in this volume for more information about sparse matrices in MuPAD 2.5.