`matexpr` User Guide

D. Bindel

January 9, 2012

1 Introduction

`matexpr` is a source-to-source translator for embedding simple MATLAB-like matrix expressions in C/C++. `matexpr` interprets specially-formatted comments in a source file and uses them to generate ordinary C code. For example, the following code computes a Rayleigh quotient for two three-by-three matrices:

```
double rayleigh_quotient3d(double* K, double* M, double* v)
{
    double rq;
    /* <generator matexpr>
     // Compute the Rayleigh quotient for a 3-by-3 pencil (K,M)

     output rq;
     input K(3,3), M(3,3), v(3);
     rq = (v'*K*v)/(v'*M*v);
    */
    return rq;
}
```

In addition to MATLAB-like matrix construction and arithmetic, `matexpr` also provides simple symbolic differentiation.

`matexpr` is *not* a full package for numerical linear algebra, nor even a particularly good substitute for a decent C++ matrix class. The purpose of `matexpr` is to make it easy to avoid index errors and unnecessary overhead when evaluating the sorts of small matrix expressions that arise in coding finite elements and other similar tasks.

2 `matexpr` command line

The `matexpr` command line has the following form:

```
matexpr [-comment] [-nogen] [-check] infile
```

where

- `-comment` specifies that `matexpr` should output labels in generated code to specify corresponding source lines. This is mostly useful for debugging generated code.
- `-line` specifies that `matexpr` should output C preprocessor `#line` labels so that error diagnostics from the C/C++ compiler will point to the appopriate place in the input file.
- `-nogen` specifies that `matexpr` should remove all automatically generated code from the output file.
- `-check` specifies that `matexpr` should check the input file without generating any other output.
- `-c99complex` specifies that `matexpr` should use C99-style complex numbers (as opposed to C++ style complex).

3 Interface syntax

The complete syntax for `matexpr` is given in Figure 1. Matrices must have known *constant* dimensions. Variables that are not explicitly declared for input or output are assumed to be scratch variables.

`matexpr` expressions are embedded in C-style comments that begin with the start-of-comment string `/* <generator matexpr>`. The starting tag can include an optional assignment of the form `complex=''`*name*`''` to specify a type to be used locally for complex inputs. The generator finishes processing at the end of the C comment. C++-style line comments may be used to document the generator code. The output of the generator is also marked off by special comments, i.e.

```
/* <generated matexpr> */ {
... Generated source goes here ...
} /* </generated> */
```

The generator will skip any code in the input file which has this form. Consequently, if `foo1.cc` is a valid input file and we run

```
matexpr foo1.cc > foo2.cc
matexpr foo2.cc > foo3.cc
```

then the files `foo2.cc` and `foo3.cc` will be identical.

4 Array handling

Matrices are represented as C arrays, but with Fortran-style column-major storage. Input arrays can be declared symmetric, in which case only the upper triangle is accessed; a matrix declared as complex and symmetric is *not* Hermitian. An array used for input or output can be specified with a leading dimension given in brackets; this is used, for example, to

```
statement := var-id = expr ;
          := var-id += expr ;
          := function id ( formals ) =  expr ;
          := iospec decls ;

iospec      := input | output | inout | complex input | complex inout
decls       := decl initializer , decl initializer , ...
decl        := var-id | var-id ( m )
            := var-id ( m , n ) | var-id symmetric ( m ) | var-id [ lda ] ( m , n )
initializer := = expr | ϵ
formals     := id , id , ...

expr        := expr : expr
            := expr + expr
            := expr - expr
            := expr * expr
            := expr / expr
            := - expr
            := expr '
            := ( expr )
            := var-id
            := number
            := matrix
            := func-id ( expr , expr , ... )
            := var-id ( expr ) | var-id ( expr , expr )
matrix      := [ rows ]
rows        := row ; row ; ...
row         := expr , expr , ...
```

Figure 1: `matexpr` call syntax

pass submatrices into `matexpr`-generated expressions. The array dimensions and the leading dimension must all be integer constants.

Expressions of the form $A(i)$ or $A(i, j)$ where A is an array are interpreted as subscript operations. At present, the subscripts *must* be compile-time integer constants. If only one index is given for a two-dimensional array, it is interpreted as the index when the entries are listed in column-major order. Indexing is one-based.

5 Functions

If `matexpr` sees an expression of the form $f(...)$, where f is not known to be a variable, it interprets the expression as a function call. If f corresponds to a declared function name, the function is called inline; if it is a special function, it is handled appropriately; and otherwise, it is interpreted as a C function call. If f is known to be a variable, the expression is interpreted as a subscript operation.

`matexpr` recognizes two special functions:

- `deriv(f, x)` – differentiate the function f with respect to the input variable(s) x. The second argument can be a matrix; for example, `deriv(f, [x, y])` is equivalent to `[ deriv(f, x), deriv(f, y) ]`. Similarly, `deriv(f, [x; y])` is equivalent to `[deriv(f,x); deriv(f,y)]`. `matexpr` only does forward-mode differentiation, and only handles basic arithmetic operations and a few elementary transcendental functions.
- `eye(n)` – produce an n-by-n identity matrix. n must be a compile time constant.

For C functions, `matexpr` currently only allows functions of one argument. If the argument specified is a matrix, `matexpr` evaluates the function elementwise.