How to use the C++ Interface to Matlab

Rahul Nair

March 20, 2009

1 Before we start (Things worth knowing)

Matlab always employs Copy on demand. In assignments of the for a = b as well as when arguments are passed to functions.Only references or pointers are passed. Only if subsequently the assigned array is changed, matlab will copy the data.

This also applies to mex functions. Matlab does not copy the data but only provides pointers to the memory, where the data is managed. Note that copy on demand is not enforced while using mexFunctions. So it is (as of version 2008) quite possible to manipulate data passed as function parameters in place. (Though most probably not really recommended, as this is not default matlab behaviour)

To avoid copying large amount of data while using the C++ Interface, but still having the comfort of using Classes and Objects, VIGRA provides View classes. These are basically enhanced pointers to external memory providing methods that work on it.

It would be advisible to have a look at the functions already written as much of the following will most probably be easier to understand.

2 The Gateway function

The mexFunction(...) gateway is replaced by the vigraMexFunction(...) gateway. The mexFunction still exists - but is in matlab.hxx. It only creates the InputArray and OutputArray objects and subsequently calls vigraMexFunction(...)

Note: if for some reason you do not want to use a custom mexFunction it is possible to override default behavior by `#define` CUSTOM_MEX_FUNCTION before `#include` matlab.hxx

The two main classes needed for using the interface are vigra::matlab::InputArray and vigra::matlab::OutputArray. These two classes take the plhs prhs and nlhs nrhs parameters of the classical mexFunction and provide methods that simplify access of data.

The job of the vigraMexFunction is to call the right template-instance of the `vigraMain<>(..)` function. Typechecking of the inputs etc should be done here.

Note: vigra contains typedefs of kind UInt8 ... UInt64, Int8 ... Int64

Listing 1: The first parameter of the Function can be of type double or of type UInt8

```
1  void vigraMexFunction(  vigra::matlab::OutputArray outputs,
2                          vigra::matlab::InputArray inputs)
3  {
4      switch(inputs.typeOf(0))
5      //switch on the type of the first parameter
6      //calls mxClassID to get the a number corresponding to the type
7      {
8      case mxUINT8_CLASS:
```

```
9           vigraMain<UInt8>(outputs, inputs);      break;
10      case mxDOUBLE_CLASS:
11          vigraMain<double>(outputs, inputs);      break;
12      default:
13          mexErrMsgTxt("Type of input at position 0 not supported");
14      }
15  }
```

Listing 2: Using Macros - This only works if vigraMain takes only one template parameter.

```
1   void vigraMexFunction(vigra::matlab::OutputArray outputs,
2                         vigra::matlab::InputArray inputs)
3   {
4       switch(inputs.typeOf("someOption"))
5       //switch on the type of the field "someOption" of the options
6       //struct.
7       //calls mxClassID to get the a number corresponding to the type
8       {
9           ALLOW_UINT_16_64     //switch cases for int16 to int64
10          ALLOW_INT_8_32       //switch cases for int8 to int32
11          ALLOW_FD          //allow float and double
12          default:
13              mexErrMsgTxt("Type of input at position 0 not supported");
14      }
15  }
```

3 The vigraMain function

The `vigraMain(...)` function contains the C++ code that actually has to be run. First the View objects should look at the right data. Also space should be allocated for the output arrays.

Listing 3: Examples

```
1   1.
2   /*Make a View of Type T (the template type of vigraMain) and let it look at the
3   first parameter supplied. v_required() indicates that an mexErrMsgTxt() is thrown
4   if no argument is supplied at position 0 (or an empty array)*/
5   BasicImageView<T>   in      =   inputs.getImage<T>(0, v_required());
6
7   2.
8   /*Second Argument is Scalar and copied into "scale" v_default(1.0) indicates that
9   if no argument is supplied the default value 1.0 is used. the last two arguments
10  are the range constraints on scale a mexErrMsgTxt is thrown if the value is out
11  of bounds*/
12  double scale   =   inputs.getScalarMinMax<double>(1, v_default(1.0), 0.0, "inf");
13
14  3.
15  //Create an Image of type double at output position 0 with the size of in
16  BasicImageView<double> out  =
17      outputs.createImage<double>(0, v_required(), in.width(), in.height());
18
19  4.
20  /*Same thing as 2. without constraints. v_optional() indicates that this variable
21  does not have a default value. v_optional(bool check) sets check to true if the
22  variable has been set, false otherwise.*/
23  bool    hasBackground;
24  T   backgroundValue =
25          inputs.getScalar<T>("backgroundValue", v_optional(hasBackground));
26
27  5.
28  /*creates a scalar at the secon output if second output has been asked for
29  (v_optional()) and copies max_region_label into it*/
30  outputs.createScalar<int> (1, v_optional(), max_region_label);
```

After space is allocated and the Views point to the right memory - the actual code can be executed.

4 matlab::InputArray and matlab::OutputArray

These are wrapper classes for plhs nlhs, prhs nrhs respectively. **InputArray checks whether the last mxArray* in prhs is a matlab struct array** - If yes it is an options struct and loaded. What followes is a listing of public methods and attributes

Place posOrName denotes an object of type std::string or an int. If it is std::string the options struct is searched for a field with name posOrNum. if it is an then the argument at the given position is used.

ReqType is one of the objects described in the next section.

Note: If you are not using matlab::InputArray or matlab::OutputArray, still you may use the non-member get and create functions which take a mxArray* as first parameter. Note that these functions do not check for constraints or whether the mxArray* is pointing to any memory. Look into matlab.hxx for further details.

Listing 4: matlab::InputArray

```
1   matlab::ConstStructArray options_
2       /*The options Struct. See Documentation of ConstStructArray for more
3       information.*/
4   mxArray* & operator[](Place posOrName)
5       /*Access reference to the mxArray* at certain place.*/
6   size_type size()
7       /*returns nrhs*/
8   bool isValid(Place posOrName)
9       /*returns true if a Argument was supplied at place posOrNum.*/
10  bool isEmpty(Place posOrName)
11      /*return true if Array at place posOrNum is empty.*/
12  mxClassID typeOf(place posOrName)
13      /*return type of mxArray* at place posOrNum;*/
14
15  template <class T,class place, class ReqType>
16  T getScalar(place posOrName, ReqType req)
17      /*get Scalar value at place posOrName. */
18
19  template<class T, class place, class reqClass,
20                                      class minClass, class maxClass>
21  T getScalarMinMax(place posOrName, reqClass req,
22                                          minClass min_, maxClass max_)
23      /*get Scalar value constrained by range defined by min_ and max_
24      min_ and max_ can also be "inf"*/
25
26  template <class T, class place, class reqClass, class iteratorType>
27  T getScalarVals(place posOrName, reqClass req,
28                                        iteratorType begin_, iteratorType end_)
29      /*get Scalar value constrained by the values in the iterator range
30      given by begin_ and end_*/
31
32  template <class T, class place, class reqClass, class iteratorType>
33  T getScalarVals2D3D(place posOrName, reqClass req,
34                                      iteratorType begin2D_, iteratorType end2D_,
35                                iteratorType begin3D_, iteratorType end3D_,
36                                                                int dimVar)
37      /*get Scalar value constrained by range begin2D_, end2D_ in cas dimVar
38      is 2 else constrained by range begin3D_, end3D_*/
39
40  template <class place, class reqClass>
41  bool getBool(place posOrName, reqClass req)
42      /*get logical value.*/
43
44
45  template <unsigned int N, class T, class place, class reqClass>
46  MultiArrayView<N,T> getMultiArray(place posOrName, reqClass req)
47      /*get MultiArrayView with dim N and Type T*/
48
49  template < class T, class place, class reqClass>
50  BasicImageView<T> getImage(place posOrName, reqClass req)
51      /*get BasicImageView with Type T*/
52
53  template<class T,unsigned int sze, class place, class reqClass>
54  TinyVectorView< T, sze> getTinyVector(place posOrName, reqClass req)
55      /*get TinyVectorView of Type T and size sze*/
```

```
56
57      template< unsigned int sze, class place, class reqClass>
58      TinyVectorView<MultiArrayIndex, sze> getShape(place posOrName, reqClass req)
59          /*get MutliarrayShape size sze*/
60
61      template< class place, class reqClass>
62      int getDimOfInput(place posOrName, reqClass req)
63          /*get Dimension of Input at place posOrName*/
64
65      template<class place, class reqClass>
66      ConstCellArray getCellArray(place posOrName, reqClass req)
67          /*get a Object of type ConstCellArray*/
```

Listing 5: matlab::OutputArray

```
1       mxArray* & operator[](int pos)
2           Access reference to the mxArray* at certain place.
3       size_type size()
4           returns nlhs
5       bool isValid(int pos)
6           returns true if a Output was required at position pos.
7       bool isEmpty(int pos)
8           return true if Array at place posOrNum is empty.
9
10      template <unsigned int DIM, class T, class ReqType>
11      MultiArrayView<DIM, T> createMultiArray(int pos,ReqType req,
12                                              const TinyVector<int, DIM> & shape)
13          /*create MultiArrayView of dimension Dim and type T and allocate
14          enough space for shape*/
15
16      template <class T, class ReqType>
17      BasicImageView<T> createImage(int pos, ReqType req,
18                                    mwSize width, mwSize height)
19
20      template <class T, class ReqType>
21      BasicImageView<T> createImage(  int pos, ReqType req,
22                                    typename MultiArrayShape<2>::type const & shape)
23
24      template <class T, class ReqType>
25      T* createScalar(int pos, ReqType req)
26          /*allocate memory for a scalar and return pointer to it.*/
27
28      template <class T, class ReqType>
29      void createScalar(int pos, ReqType req, T val)
30          /*allocate memory for a scalar and copy val into it.*/
31
32      template <class ReqType>
33      ConstCellArray createCellArray(int pos, ReqType req, mwSize sze)
```

5 The Required/Optional/Default objects

The createType and getType functions always take an object of type Required, `DefaultImpl<T>`, `DefaultImplVoid` or `bool` as second argument Use the factory methods `v_required()`, `v_default()` and `v_optional` to create these objects:

Listing 6: Behavior of the get/set functions with the factory objects

```
1       v_required(void)
2           /*generates an error message if the argument was not supplied.*/
3       v_optional()
4           /*does nothing. If argument was not supplied default
5           constructor is called.*/
6       v_optional(bool flag)
7           /*same as above. Only that flag is set if argument was supplied.
8           flag is false otherwise.*/
9
10      /*Additionally when in use with the get method:*/
11
12      v_default(defaultVal)
13          /*returns defaultVal if no argument supplied.*/
14      v_default(defaultVal2D, defaultVal3D, dimSwitch)
15          /*if dimSwitch == 2 use defaultVal2D else use defaultVal3D*/
```

Note: If you need to know whether an Object was set even if you used `v_default(defaultval)` Explicitly create the `DefaultImpl<T>` object (Only if using the first method):

Listing 7: Finding out whether a default object was set

```
//create a default MultiArray
MultiArray<3, T> in(SomeShape);
//Create defaultImpl object
DefaultImpl<typename MultiArrayView<3,T> >
                defaultMultiArrayView(MultiArrayView<3,T>(in));
MultiArrayView<3,T> Arg =
          inputs.getMultiArray<3,T>(0,v_default(defaultMultiArrayView));

//first way of checking whether Argument  was supplied:
if(defaultMultiArrayView.garbage == true)
    std::cout << "it has been set";

/*second way of checking whether Argument was supplied- only works with the
View classes, not with scalars:*/
if(in.data() == Arg.data())
    std::cout << "it has not been set";
```

6 other classes

The ConstStructArray object is used to store the options. It should not be necessary to manipulate the ConstStructArray directly.

Listing 8: ConstStructArray (Options) ConstStructArray(mxArray* matPointer) /*Constructor.*/ operator[

```
        /*access mxArray* which is stored in the StructArray with
        fieldName.*/
    isValid()
        /*is true if ConstStructArray points to a valid matPointer*/
    isValid(std::string)
        /*true if field with name specified in string exists*/
```

This is just some experimental code but quite handy if you have make and handle sparse arrays. Basically just a wrapper class to a stl::map;

Listing 9: Sparse array

```
    template<class T>
    class SparseArray
    private:
        std::map<TinyVector<int,2>, T,ShapeCmp> data;
        int width, length;
    public:
        SparseArray(int i = 1 , int j = 1)
        //calls assign;
        void assign(int i = 1, int j = 1)
        /*set intrinsic size of the SparseArray - only needed when using map
        To MxArray.*/

        T& operator()(int i, int j){
        const T get(int i, int j){
        //get and set element;
        //Any better idea? i would like to unify the get and operator()
        //functions.
        // Problem is that  operator() always passes a reference or creates
        //one.

    void mapToMxArray(mxArray * & in)
        //Creates a sparse mxArray and copies data into it.
```