IPSYS Application Programming Interface

Copyright Carnegie Mellon University

December 16, 2007

Contents

Chapter 1

Introduction

Interactive Power Systems Simulator is developed using a general programming framework modeled after other interpreted languages. The main goal of this framework is flexibility of use and extensibility. Flexibility is achieved through a scripting language covered in the User's Manual while Extensibility can be achieved either through writing scripting or hard coded functions. Writing scripts and scripting functions is explained in the User's Manual and this manual describes how to add new hard coded procedures.

Hard coded functions can be added to the interpreter by following a well defined series of steps. Each function must input and output data using two lists of pointers to C++ Data objects and return a pointer to a single data object. A function satisfying these minimal requirements can be declared in a framework's header file with other and entered in a lookup table. In addition to these minimal requirements, to be useful, each new function must check all the input Data objects for type and value validity and either report any errors or do the required calculations and return the results. New functions can operate and return data objects representing real (double) numbers, matrices, and text strings. These data types are sufficient for any numerical and almost any other processing.

It should be noted that much more time was spent on design concepts than programming new functions. Some of the features were not explored due to time constraints and will not be discussed in this manual. Hopefully, there will be enough time and interest to continue with the development.

Chapter 2

Data Types

One of the interpreter goals is type free data. This was achieved by using a class Data with a number of pure virtual functions and operators. From this base class Real, String, and Matrix classes are derived for which all purely virtual base functions and operators had to be defined for all data type combinations. If an operator is not defined for a pair of data objects, an exception is thrown. The base class, Data, carries information about the variable type, name, and garbage collecting accounting data. For adding new hard coded functions, the two things to remember are that Data class has no functionality and there is no reason to instantiate it and that the derived classes cannot be instantiated on the stack. The nice thing about this setup is that the garbage collection is automatic and interpreter and hard coded functions can have well defined data scopes. Once the interpreter is done parsing a complete statement, all of the temporary and/or lower scope data objects are deleted. All unnamed data within a scope is considered temporary and is deleted after each complete statement or an error.

The only way to create a data object is by calling create function, the following methods can be used to create Real objects:

```
static Real &create(double dbl, const std::string & = std::string(""))
   throw(std::bad_alloc);
static Real &create(double dbl, char *nm) throw(std::bad_alloc);
static Real &create() throw(std::bad_alloc);
static Real &create(const Real &, const std::string &
   = std::string("")) throw(std::bad_alloc);
```

The definitions of data classes can be found in Data.hpp, Real.hpp, Matrix.hpp, and String.hpp and should be consulted for all of the data and methods definitions. Operations and functions that can be used with these data types are the same ones described in The User's Manual and any newly defined functions. For example, the following code will define and add a Real and a Matrix:

```
Real &rlp = Real::create(13.13);
TNT::Matrix<double> a(4,3,0.0);
a(1,1) = 0.0; a(1,2) = 3; a(1,3) = 1;
a(2,1) = 3; a(2,2) = 1; a(2,3) = 2;
a(3,1) = -12; a(3,2) = -9; a(3,3) = 8;
a(4,1) = -1; a(4,2) = 1; a(4,3) = 13;
Matrix &A = Matrix::create(a);
std::cout << "A + rlp = " << (A+rlp) << std::endl;

A + rlp = 13.13 16.13 14.13
16.13 14.13 15.13
1.13 4.13 21.13
12.13 14.13 26.13
```

This should be a sufficient introduction to the data types used by the framework. The corresponding header files should be used as the ultimate reference.

Chapter 3

Functions, Input/Output, and Error Handling

Writing functions and adding functions to the framework correctly is all that is needed to add new functionality to the interpreter. Each function must be defined for example as:

```
Data *Cos(std::vector<Data *> &in, std::vector<Data *> &out);
```

The vector in is the vector of pointers to Data objects which can actually be Real, Matrix, or String since Data is the base class. in is the input data and the number of input Data objects should be checked by checking in.size() value. out is the vector of output Data object pointers and it should always be cleared with out.clear() before pushing any values into it since the interpreter will pass this vector to the functions and might already contain some data. The return Data pointer should always be the out[0] value to match the expected interpreter behavior.

Once the function is defined, it needs to be entered in the lookup table:

```
 __cppFuncs["cos"] = Cos;
```

The code segment above means that function Cos(...) will be recognized by the interpreter as function cos(....). Definition of __cppFuncs can be found in ipsys.hpp as:

```
typedef Data *(*CPPFunc)(std::vector<Data *> &, std::vector<Data *> &);
extern std::map<std::string, CPPFunc, std::less<std::string> > __cppFuncs;
```

If the function is defined as described above, the source file can be added to be compiled with the rest of the program and it will be recognized and used by the interpreter. However, whether it does or not what it is intended to do is another question. First, the function must check the input arguments and if there is something wrong throw an exception. There is no need to clean up created objects or actually catch the thrown exception. The interpreter will clean up the temporary objects and catch the thrown exception. If all of the input data is correct, the calculations can be done, return objects created and pushed into out vector. out[0] should be function's return value. Note that something must be returned. The complete implementation of the above function is taken from ipsys code:

```
typedef double (*func)(double);

Data *applyFunc(Data *dtp, func f)
{

  if(dtp->type_ == REAL)
  {
    Real *rdtp = &Real::create(0);
    dynamic_cast<Real *>(rdtp)->real_ = f(dynamic_cast<Real *>(dtp)->real_);
    return rdtp;
  }
  else if(dtp->type_ == MATRIX)
  {
    const int r = dynamic_cast<Matrix *>(dtp)->matrx_.num_rows();
    const int c = dynamic_cast<Matrix *>(dtp)->matrx_.num_cols();
    Matrix *rdtp = &Matrix::create(TNT::Matrix<double>(r,c,0.0));
    for(int i = 1; i <= r; i++)
    {
      for(int j = 1; j <= c; j++)
      {
        dynamic_cast<Matrix *>(rdtp)->matrx_(i,j) =
        f(dynamic_cast<Matrix *>(dtp)->matrx_(i,j));
      }
    }
    return rdtp;
  }
```

```
  else
    throw DataError("Illegal data type");

}

Data *Cos(std::vector<Data *> &in, std::vector<Data *> &out) throw(DataError)
{

  if(in.size() != 1 )
    throw DataError("Incorrent number of arguments");

  out.clear();
  out.push_back(applyFunc(in[0],cos));

  return out[0];
}
```

This is all that was done in IPSYS to add the cosine function. Using exactly
the same code and replacing cos with another one argument math function
was used to add other math functions. After becoming familiar with the
requirements and what the framework provides, adding new, even most com-
plex functions, becomes a routine. The restrictions can come from the native
language itself and/or the operating system. Considering that C++ can link
C and F77 libraries the language limits are quite lax. The development of
the entire package was done under Windows and at some points there were
doubts where the bugs were originating. There is no portability reason why
the entire package could not be compiled under Unix although the more
recent GNU g++ compiler versions have rather interesting behavior.

Chapter 4

Bugs

There are no documented bugs related to use of the API but they are bound
to surface with more use.

Chapter 5

To Do

At present, addition of any hard coded functions require linking them with the entire application. Ideally, they should be imported through dynamic linking without relinking the entire program. Windows operating system is not as friendly as Unix to such dynamic linking.