Chapter 4. Data Types

Authors: *Joseph T. Buck*
Michael J. Chen
Alireza Khazeni

Other Contributors: *Brian Evans*
Paul Haskell
Asawaree Kalavade
Tom Lane
Edward A. Lee
John Reekie

4.1 Introduction

Stars communicate by sending objects of type `Particle`. A basic set of types, including scalar and array types, built on the `Particle` class, is built into the Ptolemy kernel. Since all of these particle types are derived from the same base class, it is possible to write stars that operate on any of them (by referring only to the base class). It is also possible to define new types that contain arbitrary C++ objects.

There are currently eleven key data types defined in the Ptolemy kernel. There are four numeric scalar types—complex, fixed-point, double precision floating-point, and integer—described in Section 4.2. Ptolemy supports a limited form of user-defined type—the *Message* type, described in Section 4.3. Each of the scalar numeric types has an equivalent matrix type, which uses a more complex version of the user-defined type mechanism; they are described in Section 4.4.

There are two experimental types included in the basic set, containing strings and file references, described in Section 4.5. Ptolemy allows stars to be written that will read and write particles of any type; this mechanism is described in Section 4.6. Finally, some experimental types that are not officially supported by Ptolemy are described in Section 4.7.

4.2 Scalar Numeric Types

There are four scalar numeric data types defined in the Ptolemy kernel: complex, fixed-point, double precision floating-point, and integer. All of these four types can be read from and written to portholes as described in “Reading inputs and writing outputs” on page 2-17. The floating-point and integer data types are based on the standard C++ `double` and `int` types, and need no further explanation. To support the other two types, the Ptolemy kernel contains a `Complex` class and a `Fix` class, which are described in the rest of this section.

4.2.1 The Complex data type

The `Complex` data type in Ptolemy contains real and imaginary components, each of which is specified as a double precision floating-point number. The notation used to represent a complex number is a two number pair: (real, imaginary)—for example, (1.3,-4.5) corresponds to the complex number $1.3 - 4.5j$. `Complex` implements a subset of the functionality of the complex number classes in the cfront and libg++ libraries, including most of the standard arithmetic operators and a few transcendental functions.

Constructors:

`Complex()`

Create a complex number initialized to zero—that is, (0.0, 0.0). For example,

`Complex C;`

`Complex(double real, double imag)`

Create a complex number whose value is (real, imag). For example,

`Complex C(1.3,-4.5);`

`Complex(const Complex& arg)`

Create a complex number with the same value as the argument (the copy constructor). For example,

`Complex A(complexSourceNumber);`

Basic operators:

The following list of arithmetic operators modify the value of the complex number. All functions return a reference to the modified complex number (`*this`).

```
Complex& operator = (const Complex& arg)

Complex& operator += (const Complex& arg)

Complex& operator -= (const Complex& arg)

Complex& operator *= (const Complex& arg)

Complex& operator /= (const Complex& arg)

Complex& operator *= (double arg)

Complex& operator /= (double arg)
```

There are two operators to return the real and imaginary parts of the complex number:

```
double real() const

double imag() const
```

Non-member functions and operators:

The following one- and two-argument operators return a new complex number:

```
Complex operator + (const Complex& x, const Complex& y)

Complex operator - (const Complex& x, const Complex& y)

Complex operator * (const Complex& x, const Complex& y)
```

```
Complex operator * (double x, const Complex& y)

Complex operator * (const Complex& x, double y)

Complex operator / (const Complex& x, const Complex& y)

Complex operator / (const Complex& x, double y)

Complex operator - (const Complex& x)
```
Return the negative of the complex number.

```
Complex conj (const Complex& x)
```
Return the complex conjugate of the number.

```
Complex sin(const Complex& x)

Complex cos(const Complex& x)

Complex exp(const Complex& x)

Complex log(const Complex& x)

Complex sqrt(const Complex& x)

Complex pow(double base, const Complex& expon)

Complex pow(const Complex& base, const Complex& expon)
```

Other general operators:

```
double abs(const Complex& x)
```
Return the absolute value, defined to be the square root of the norm.

```
double arg(const Complex& x)
```
Return the value arctan(x.imag()/x.real()).

```
double norm(const Complex& x)
```
Return the value x.real() * x.real() + x.imag() * x.imag().

```
double real(const Complex& x)
```
Return the real part of the complex number.

```
double imag(const Complex& x)
```
Return the imaginary part of the complex number.

Comparison Operators:

```
int operator != (const Complex& x, const Complex& y)

int operator == (const Complex& x, const Complex& y)
```

4.2.2 The fixed-point data type

The fixed-point data type is implemented in Ptolemy by the `Fix` class. This class supports a two's complement representation of a finite precision number. In fixed-point notation, the partition between the integer part and the fractional part—the binary point—lies at a fixed position in the bit pattern. Its position represents a trade-off between precision and range. If the binary point lies to the right of all bits, then there is no fractional part.

Constructing Fixed-point variables

Variables of type `Fix` are defined by specifying the word length and the position of the binary point. At the user-interface level, precision is specified either by setting a fixed-point parameter to a "(*value*, *precision*)" pair, or by setting a `precision` parameter. The former gives the value and precision of some fixed-point value, while the latter is typically used to specify the internal precision of computations in a star.

In either case, the syntax of the precision is either *"x.y"* or "*m*/*n*", where *x* is the number of integer bits (including the sign bit), *y* and *m* are the number of fractional bits, and *n* is the total number of bits. Thus, the total number of bits in the fixed-point number (also called its *length*) is *x*+*y* or *n*. For example, a fixed-point number with precision "3.5" has a total length of 8 bits, with 3 bits to the left and 5 bits to the right of the binary point.

At the source code level, methods working on `Fix` objects either have the precision passed as an "*x.y*" or "*m*/*n*" string, or as two C++ integers that specify the total number of bits and the number of integer bits including the sign bit (that is, *n* and *x*). For example, suppose you have a star with a precision parameter named *precision*. Consider the following code:

```
Fix x = Fix(((const char *) precision));
if (x.invalid())
       Error::abortRun(*this, "Invalid precision");
```

The "precision" parameter is cast to a string and passed as a constructor argument to the `Fix` class. The error check verifies that the precision was valid.

There is a maximum value for the total length of a `Fix` object which is specified by the constant `FIX_MAX_LENGTH` in the file `$PTOLEMY/src/kernel/Fix.h`. The current value is 64 bits. Numbers in the `Fix` class are represented using two's complement notation, with the sign bit stored in the bits to the left of the binary point. There must always be at least one bit to the left of the binary point to store the sign.

In addition to its value, each `Fix` object contains information about its precision and error codes indicating overflow, divide-by-zero, or bad format parameters. The error codes are set when errors occur in constructors or arithmetic operators. There are also fields to specify

a. whether rounding or truncation should take place when other `Fix` values are assigned to it—truncation is the default

b. the response to an overflow or underflow on assignment—the default is saturation (see page 4-6).

Warning

The `Fix` type is still experimental.

Fixed-point states

State variables can be declared as `Fix` or `FixArray`. The precision is specified by an associated precision state using either of two syntaxes:

- Specifying just a value itself in the dialog box creates a fixed-point number with the default length of 24 bits and with the position of the binary point set as required to store the integer value. For example, the value 1.0 creates a fixed-point object with precision 2.22, and the value 0.5 would create one with precision 1.23.
- Specifying a (value, precision) pair create a fixed-point number with the specified precision. For example, the value (2.546, 3.5) creates a fixed-point object by casting the double 2.546 to a `Fix` with precision 3.5.

Fixed-point inputs and outputs

`Fix` types are available in Ptolemy as a type of `Particle`. The conversion from an `int` or a `double` to a `Fix` takes place using the `Fix::Fix(double)` constructor which makes a `Fix` object with the default word length of 24 bits and the number of integer bits as needed required by the value. For instance, the `double` 10.3 will be converted to a `Fix` with precision 5.19, since 5 is the minimum number of bits needed to represent the integer part, 10, including its sign bit.

To use the `Fix` type in a star, the type of the portholes must be declared as “`fix`”. Stars that receive or transmit fixed-point data have parameters that specify the precision of the input and output in bits, as well as the overflow behavior. Here is a simplified version of `SDFAddFix` star, configured for two inputs:

```
defstar {
      name { AddFix }
      domain {SDF}
      derivedFrom{ SDFFix }
      input {
            name { input1 }
            type { fix }
      }
      input {
            name { input2 }
            type { fix }
      }
      output {
            name { output }
            type { fix }
      }
      defstate {
            name { OutputPrecision }
            type { precision }
            default { 2.14 }
      desc {
Precision of the output in bits and precision of the accumulation.
When the value of the accumulation extends outside of the precision,
the OverflowHandler will be called.
      }
}
```

(Note that the real `AddFix` star supports any number of inputs.) By default, the precision used by this star during the addition will have 2 bits to the left of the binary point and 14 bits to the right. Not shown here is the state `OverflowHandler`, which is inherited from the `SDFFix` star and which defaults to `saturate`—that is, if the addition overflows, then the result saturates, pegging it to either the largest positive or negative number representable. The result value, `sum`, is initialized by the following code:

```
protected {
      Fix sum;
}
begin {
      SDFFix::begin();

      sum = Fix( ((const char *) OutputPrecision) );
      if ( sum.invalid() )
            Error::abortRun(*this, "Invalid OutputPrecision");
      sum.set_ovflow( ((const char*) OverflowHandler) );
      if ( sum.invalid() )
            Error::abortRun(*this, "Invalid OverflowHandler");
}
```

The `begin` method checks the specified precision and overflow handler for correctness. Then, in the `go` method, we use `sum` to calculate the result value, thus guaranteeing that the desired precision and overflow handling are enforced. For example,

```
go {
      sum.setToZero();
      sum += Fix(input1%0);
      checkOverflow(sum);
      sum += Fix(input2%0);
      checkOverflow(sum);
      output%0 << sum;
}
```

(The `checkOverflow` method is inherited from `SDFFix`.) The protected member `sum` is an *uninitialized* `Fix` object until the `begin` method runs. In the `begin` method, it is given the precision specified by `OutputPrecision`. The `go` method initializes it to zero. If the `go` method had instead assigned it a value specified by another `Fix` object, then it would acquire the precision of that other object—at that point, it would be *initialized*.

Assignment and overflow handling

Once a `Fix` object has been initialized, its precision does not change as long as the object exists. The assignment operator is overloaded so that it checks whether the value of the object to the right of the assignment fits into the precision of the left object. If not, then it takes the appropriate overflow response is taken and set the overflow error bit.

If a `Fix` object is created using the constructor that takes no arguments, as in the `protected` declaration above, then that object is an uninitialized `Fix`; it can accept any assignment, acquiring not only its value, but also its precision and overflow handler.

The behavior of a `Fix` object on an overflow depends on the specifications and the behavior of the object itself. Each object has a private data field that is initialized by the constructor; when there is an overflow, the `overflow_handler` looks at this field and uses the specified method to handle the overflow. This data field is set to `saturate` by default, and can be set explicitly to any other desired overflow handling method using a function called `set_ovflow(<keyword>)`. The keywords for overflow handling methods are: `saturate` (default), `zero_saturate`, `wrapped`, `warning`. `saturate` replaces the original value is replaced by the maximum (for overflow) or minimum (for underflow) value representable given the precision of the `Fix` object. `zero_saturate` sets the value to zero.

Explicitly casting inputs

In the above example, the first line of the `go` method assigned the input to the protected member `sum`, which has the side-effect of quantizing the input to the precision of `sum`. We could have alternatively written the `go` method as follows:

```
go {
      sum = Fix(input1%0) + Fix(input2%0);
      output%0 << sum;
}
```

The behavior here is significantly different: the inputs are added using their own native precision, and only the result is quantized to the precision of `sum`.

Some stars allow the user to select between these two different behaviors with a parameter called *ArrivingPrecision*. If set to `YES`, the input particles are not explicitly cast; they are used as they are; if set to `NO`, the input particles are cast to an internal precision, which is usually specified by another parameter.

Here is the (abbreviated) source of the `SDFGainFix` star, which demonstrates this point:

```
defstar {
      name { GainFix }
      domain { SDF }
      derivedFrom { SDFFix }
      desc {
This is an amplifier; the fixed-point output is the fixed-point input
multiplied by the "gain" (default 1.0). The precision of "gain", the
input, and the output can be specified in bits.
      }
      input {
            name { input }
            type { fix }
      }
      output {
            name { output }
            type { fix }
      }
      defstate {
            name { gain }
            type { fix }
            default { 1.0 }
            desc { Gain of the star. }
      }
```

```
        defstate {
              name { ArrivingPrecision }
              type {int}
              default {"YES"}
              desc {
  Flag indicating whether or no to use the arriving particles as they
  are: YES keeps the same precision, and NO casts them to the precision
  specified by the parameter "InputPrecision". }
        }
        defstate {
              name { InputPrecision }
              type { precision }
              default { 2.14 }
              desc {
  Precision of the input in bits. The input particles are only cast
  to this precision if the parameter "ArrivingPrecision" is set to NO.
              }
        }
        defstate {
              name { OutputPrecision }
              type { precision }
              default { 2.14 }
              desc {
  Precision of the output in bits.
  This is the precision that will hold the result of the arithmetic
  operation on the inputs.
  When the value of the product extends outside of the precision,
  the OverflowHandler will be called.
        }
        protected {
              Fix fixIn, out;
        }
        begin {
              SDFFix::begin();

              if ( ! int(ArrivingPrecision) ) {
                 fixIn = Fix( ((const char *) InputPrecision) );
                 if(fixIn.invalid())
                    Error::abortRun( *this, "Invalid InputPrecision" );
              }

              out = Fix( ((const char *) OutputPrecision) );
              if ( out.invalid() )
                 Error::abortRun( *this, "Invalid OutputPrecision" );
              out.set_ovflow( ((const char *) OverflowHandler) );
                 if(out.invalid())
                    Error::abortRun( *this,"Invalid OverflowHandler" );
        }
        go {
              // all computations should be performed with out since
              // that is the Fix variable with the desired overflow
              // handler
              out = Fix(gain);
              if ( int(ArrivingPrecision) ) {
```

```
                        out *= Fix(input%0);
                }
                else {
                        fixIn = Fix(input%0);
                        out *= fixIn;
                }
                checkOverflow(out);
                output%0 << out;
        }
        // a wrap-up method is inherited from SDFFix
        // if you defined your own, you should call SDFFix::wrapup()
}
```

Note that the `SDFGainFix` star and many of the `Fix` stars are derived from the star `SDFFix`. `SDFFix` implements commonly used methods and defines two states: `OverflowHandler` selects one of four overflow handlers to be called each time an overflow occurs; and `ReportOverflow`, which, if true, causes the number and percentage of overflows that occurred for that star during a simulation run to be reported in the `wrapup` method.

Constructors:

`Fix()` Create a `Fix` number with unspecified precision and value zero.

`Fix(int length, int intbits)`
: Create a `Fix` number with total word length of `length` bits and `intbits` bits to the left of the binary point. The value is set to zero. If the precision parameters are not valid, then an error bit is internally set so that the `invalid` method will return `TRUE`.

`Fix(const char* precisionString)`
: Create a `Fix` number whose precision is determined by `precisionString`, which has the syntax "*leftbits.rightbits*", where *leftbits* is the number of bits to the left of the binary point and *rightbits* is the number of bits to the right of the binary point, or "*rightbits/totalbits*", where *totalbits* is the total number of bits. The value is set to zero. If the `precisionString` is not in the proper format, an error bit is internally set so that the `invalid` method will return `TRUE`.

`Fix(double value)`
: Create a `Fix` with the default precision of 24 total bits for the word length and set the number of integer bits to the minimum needed to represent the integer part of the number value. If the value given needs more than 24 bits to represent, the value will be clipped and the number stored will be the largest possible under the default precision (i.e. saturation occurs). In this case an internal error bit is set so that the `ovf_occurred` method will return `TRUE`.

`Fix(int length, int intbits, double value)`
: Create a `Fix` with the specified precision and set its value to the given `value`. The number is rounded to the closest representable number

given the precision. If the precision parameters are not valid, then an error bit is internally set so that the `invalid` method will return `TRUE`.

`Fix(const char* precisionString, double value)`
Same as the previous constructor except that the precision is specified by the given `precisionString` instead of as two integer arguments. If the precision parameters are not valid, then an error bit is internally set so that the `invalid()` method will return true when called on the object.

`Fix(const char* precisionString, uint16* bits)`
Create a `Fix` with the specified precision and set the bits precisely to the ones in the given `bits`. The first word pointed to by `bits` contains the most significant 16 bits of the representation. Only as many words as are necessary to fetch the bits will be referenced from the `bits` argument. For example: `Fix("2.14",bits)` will only reference `bits[0]`.

This constructor gets very close to the representation and is meant mainly for debugging. It may be removed in the future.

`Fix(const Fix& arg)`
Copy constructor. Produces an exact duplicate of `arg`.

`Fix(int length, int intbits, const Fix& arg)`
Read the value from the `Fix` argument and set to a new precision. If the precision parameters are not valid, then an error bit is internally set so that the `invalid` method will return true when called on the object. If the value from the source will not fit, an error bit is set so that the `ovf_occurred` method will return `TRUE`.

Functions to set or display information about the Fix number:

`int len() const`
Return the total word length of the Fix number.

`int intb() const`
Return the number of bits to the left of the binary point.

`int precision() const`
Return the number of bits to the right of the binary point.

`int overflow() const`
Return the code of the type of overflow response for the `Fix` number. The possible codes are:
`0 - ovf_saturate`,
`1 - ovf_zero_saturate`,
`2 - ovf_wrapped`,
`3 - ovf_warning`,
`4 - ovf_n_types`.

`int roundMode() const`
: Return the rounding mode: `1` for rounding, `0` for truncation.

`int signBit() const`
: Return `TRUE` if the value of the `Fix` number is negative, `FALSE` if it is positive or zero.

`int is_zero()`
: Return `TRUE` if the value of the `Fix` number is zero.

`double max()`
: Return the maximum value representable using the current precision.

`double min()`
: Return the minimum value representable using the current precision.

`double value()`
: The value of the `Fix` number as a double.

`void setToZero()`
: Set the value of the `Fix` number to zero.

`void set_overflow(int value)`
: Set the overflow type.

`void set_rounding(int value)`
: Set the rounding type: `TRUE` for rounding, `FALSE` for truncation.

`void initialize()`
: Discard the current precision format and set the `Fix` number to zero.

There are a few functions for backward compatibility:

`void set_ovflow(const char*)`
: Set the overflow using a name.

`void Set_MASK(int value)`
: Set the rounding type. Same functionality as `set_rounding()`.

Comparison function:

`int compare (const Fix& a, const Fix& b)`
: Compare two `Fix` numbers. Return –1 if $a < b$, 0 if $a = b$, 1 if $a > b$.

The following functions are for use with the error condition fields:

`int ovf_occurred()`
: Return `TRUE` if an overflow has occurred as the result of some operation like addition or assignment.

`int invalid()`
: Return `TRUE` if the current value of the `Fix` number is invalid due to it having an improper precision format, or if some operation caused a

divide by zero.

`int dbz()` Return `TRUE` if a divide by zero error occurred.

`void clear_errors()`
Reset all error bit fields to zero.

Operators:

`Fix& operator = (const Fix& arg)`
Assignment operator. If `*this` does not have its precision format set (i.e. it is uninitialized), the source `Fix` is copied. Otherwise, the source `Fix` value is converted to the existing precision. Either truncation or rounding takes place, based on the value of the rounding bit of the current object. Overflow results either in saturation, "zero saturation" (replacing the result with zero), or a warning error message, depending on the overflow field of the object. In these cases, `ovf_occurred` will return `TRUE` on the result.

`Fix& operator = (double arg)`
Assignment operator. The double value is first converted to a default precision `Fix` number and then assigned to `*this`.

The function of these arithmetic operators should be self-explanatory:

```
Fix& operator += (const Fix&)

Fix& operator -= (const Fix&)

Fix& operator *= (const Fix&)

Fix& operator *= (int)

Fix& operator /= (const Fix&)

Fix operator + (const Fix&, const Fix&)

Fix operator - (const Fix&, const Fix&)

Fix operator * (const Fix&, const Fix&)

Fix operator * (const Fix&, int)

Fix operator * (int, const Fix&)

Fix operator / (const Fix&, const Fix&)

Fix operator - (const Fix&) // unary minus

int operator == (const Fix& a, const Fix& b)

int operator != (const Fix& a, const Fix& b)

int operator >= (const Fix& a, const Fix& b)

int operator <= (const Fix& a, const Fix& b)

int operator > (const Fix& a, const Fix& b)
```

```
int operator < (const Fix& a, const Fix& b)
```

Note:

- These operators are designed so that overflow does not, as a rule, occur (the return value has a wider format than that of its arguments). The exception is when the result cannot be represented in a `Fix` with all 64 bits before the binary point.
- The output of any operation will have error codes that are the logical OR of those of the arguments to the operation, plus any additional errors that occurred during the operation (like divide by zero).
- The division operation is currently a cheat: it converts to double and computes the result, converting back to `Fix`.
- The relational operators ==, !=, >=, <=, >, < are all written in terms of a function
  `int compare(const Fix& a, const Fix& b)`
  This functions returns -1 if $a < b$, 0 if $a = b$, and 1 if $a > b$. The comparison is exact (every bit is checked) if the two values have the same precision format. If the precisions are different, the arguments are converted to doubles and compared. Since `double` values only have an accuracy of about 53 bits on most machines, this may cause false equality reports for `Fix` values with many bits.

Conversions:

`operator int() const`

Return the value of the `Fix` number as an integer, truncating towards zero.

`operator float() const`

`operator double() const`

Convert to a float or a double, creating an exact result when possible.

`void complement()`

Replace the current value by its complement.

Fix overflow, rounding, and errors.

The `Fix` class defines the following enumerated values for overflow handling:

`Fix::ovf_saturate`

`Fix::ovf_zero_saturate`

`Fix::ovf_wrapped`

`Fix::ovf_warning`

They may be used as arguments to the `set_overflow` method, as in the following example:

```
out.set_overflow(Fix::ovf_saturate);
```

The member function

```
int overflow() const;
```

returns the overflow type. This returned result can be compared against the above enumerated values. Overflow types may also be specified as strings, using the method

```
void set_ovflow(const char* overflow_type);
```

the `overflow_type` argument may be one of `saturate`, `zero_saturate`, `wrapped`, or `warning`.

The rounding behavior of a `Fix` value may be set by calling

```
void set_rounding(int value);
```

If the argument is false, or has the value `Fix::mask_truncate`, truncation will occur. If the argument is nonzero (for example, if it has the value `Fix::mask_truncate_round`, rounding will occur. The older name `Set_MASK` is a synonym for `set_rounding`.

The following functions access the error bits of a `Fix` result:

```
int ovf_occurred() const;
int invalid() const;
int dbz() const;
```

The first function returns `TRUE` if there have been any overflows in computing the value. The second returns `TRUE` if the value is invalid, because of invalid precision parameters or a divide by zero. The third returns `TRUE` only for divide by zero.

4.3 Defining New Data Types

The Ptolemy heterogeneous message interface provides a mechanism for stars to transmit arbitrary objects to other stars. Our design satisfies the following requirements:

- Existing stars (stars that were written before the message interface was added) that handle `ANYTYPE` work with message particles without change.
- Message portholes can send different types of messages during the same simulation. This is especially useful for modeling communication networks.
- It avoids copying large messages by using a reference count mechanism, as in many C++ classes (for example, string classes).
- It is possible to safely modify large messages without excessive memory allocation and deallocation.
- It is (relatively) easy for users to define their own message types; no change to the kernel is required to support new message types.

The "message" type is understood by Ptolemy to mean a particle containing a message. There are three classes that implement the support for message types:

- The `Message` class is the base class from which all other message data types are derived. A user who wishes to define an application-specific message type derives a new class from `Message`.
- The `Envelope` class contains a pointer to an derived from `Message`. When an `Envelope` objects is copied or duplicated, the new envelope simply sets its own pointer to

the pointer contained in the original. Several envelopes can thus reference the same Message object. Each Message object contains a reference count, which tracks how many Envelope objects reference it; when the last reference is removed, the Message is deleted.

- The MessageParticle class is a type of Particle (like IntParticle, FloatParticle, etc.); it contains a Envelope. Ports of type "message" transmit and receive objects of this type.

Class Particle contains two member functions for message support: getMessage, to receive a message, and the << operator with an Envelope as the right argument, to load a message into a particle. These functions return errors in the base class; they are overridden in the MessageParticle class with functions that perform the expected operation.

4.3.1 Defining a new Message class

Every user-defined message is derived from class Message. Certain virtual functions defined in that class must be overridden; others may optionally be overridden. Here is an example of a user-defined message type:

```
// This is a simple vector message object. It stores
// an array of integer values of arbitrary length.
// The length is specified by the constructor.
#include "Message.h"
class IntVecData: public Message {
private:
      int len;
      init(int length,int *srcData) {
            len = length;
            data = new int[len];
            for (int i = 0; i < len; i++)
                  data[i] = *srcData++;
      }
public:
      // the pointer is public for simplicity
      int *data;

      int length() const { return len;}

      // functions for type-checking
      const char* dataType() const { return "IntVecData";}
      // isA responds TRUE if given the name of the class or
      // of any baseclass.
      int isA(const char* typ) const {
            if (strcmp(typ,"IntVecData") == 0) return TRUE;
            else return Message::isA(typ);
      }
      // constructor: makes an uninitialized array
      IntVecData(int length): len(length) {
            data = new int[length];
      }
      // constructor: makes an initialized array from a int array
      IntVecData(int length,int *srcData) { init(length,srcData);}
```

```
        // copy constructor
        IntVecData(const IntVecData& src) { init(src.len,src.data);}

        // clone: make a duplicate object
        Message* clone() const { return new IntVecData(*this);}

        // destructor
        ~IntVecData() {
              delete data;
        }
};
```

This message object can contain a vector of integers of arbitrary length. Some functions in the class are arbitrary and the user may define them in whatever way is most convenient; however, there are some requirements.

The class must redefine the `dataType` method from class `Message`. This function returns a string identifying the message type. This string should be identical to the name of the class. In addition, the `isA` method must be defined. The `isA` method responds with `TRUE` (1) if given the name of the class or of any base class; it returns `FALSE` (0) otherwise. This mechanism permits stars to handle any of a whole group of message types, even for classes that are defined after the star is written.

Because of the regular structure of `isA` function bodies, macros are provided to generate them. The `ISA_INLINE` macro expands to an inline definition of the function; for example,

```
ISA_INLINE(IntVecData,Message)
```

could have been written above instead of the definition of `isA` to generate exactly the same code. Alternatively, to put the function body in a `.cc` file, one can write

```
int isA(const char*) const;
```

in the class definition and put

```
ISA_FUNC(IntVecData,Message)
```

in the `.cc` file (or wherever the methods are defined).

The class must define a copy constructor, unless the default copy constructor generated by the compiler, which does memberwise copying, will do the job.

The class must redefine the `clone` method of class `Message`. Given that the copy constructor is defined, the form shown in the example, where a new object is created with the `new` operator and the copy constructor, will suffice.

In addition, the user may optionally define type conversion and printing functions if they make sense. If a star that produces messages is connected to a star that expects integers (or floating values, or complex values), the appropriate type conversion function is called. The base class, `Message`, defines the virtual conversion functions `asInt()`, `asFloat()`, and `asComplex()` and the printing method `print()` — see the file `$PTOLEMY/src/kernel/Message.h` for their exact types. The base class conversion functions assert a run-time error, and the default print function returns a `StringList` saying

<*type*>: no print method

where *type* is whatever is returned by `dataType()`.

By redefining these methods, you can make it legal to connect a star that generates messages to a star that expects integer, floating, or complex particles, or you can connect to a `Printer` or `XMgraph` star (for `XMgraph` to work, you must define the `asFloat` function; for `Printer` to work, you must define the `print` method).

4.3.2 Use of the Envelope class

The `Envelope` class references objects of class `Message` or derived classes. Once a message object is placed into an envelope object, the envelope takes over responsibility for managing its memory: maintaining reference counts, and deleting the message when it is no longer needed.

The constructor (which takes as its argument a reference to a `Message`), copy constructor, assignment operator, and destructor of `Envelope` manipulate the reference counts of the references `Message` object. Assignment simply copies a pointer and increments the reference count. When the destructor of a `Envelope` is called, the reference count of the `Message` object is decremented; if it becomes zero, the `Message` object is deleted. Because of this deletion, a `Message` must never be put inside a `Envelope` unless it was created with the `new` operator. Once a `Message` object is put into an `Envelope` it must never be explicitly deleted; it will "live" as long as there is at least one `Envelope` that contains it, and it will then be deleted automatically.

It is possible for an `Envelope` to be "empty". If it is, the `empty` method will return `TRUE`, and the data field will point to a special "dummy message" with type `DUMMY` that has no data in it.

The `dataType` method of `Envelope` returns the datatype of the contained `Message` object; the methods `asInt()`, `asFloat()`, `asComplex()`, and `print()` are also "passed through" in a similar way to the contained object.

Two `Envelope` methods are provided for convenience to make type checking simpler: `typeCheck` and `typeError`. A simple example illustrates their use:

```
if (!envelope.typeCheck("IntVecData")) {
      Error::abortRun(*this, envelope.typeError("IntVecData"));
      return;
}
```

The method `typeCheck` calls `isA` on the message contents and returns the result, so an error will be reported if the message contents are not `IntVecData` and are not derived from `IntVecData`. Since the above code segment is so common in stars; a macro is included in `Message.h` to generate it; the macro

```
TYPE_CHECK(envelope,"IntVecData");
```

expands to essentially the same code as above. The `typeError` method generates an appropriate error message:

`Expected message type 'arg', got 'type'`

To access the data, two methods are provided: `myData()` and `writableCopy()`. The `myData` function returns a pointer to the contained `Message`-derived object. *The data pointed* *to by this pointer must not be modified*, since other `Envelope` objects in the program may also contain it. If you convert its type, always make sure that the converted type is a pointer to `const` (see the programming example for `UnPackInt` below). This ensures that the compiler will complain if you do anything illegal.

The `writableCopy` function also returns a pointer to the contained object, but with a difference. If the reference count is one, the envelope is emptied (set to the dummy message) and the contents are returned. If the reference count is greater than one, a *clone* of the contents is made (by calling its `clone()` function) and returned; again the envelope is zeroed (to prevent the making of additional clones later on).

In some cases, a star writer will need to keep a received `Message` object around between executions. The best way to do this is to have the star contain a member of type `Envelope`, and to use this member object to hold the message data between executions. Messages should always be kept in envelopes so that the user does not have to worry about managing their memory.

4.3.3 Use of the MessageParticle class

If a porthole is of type "message", then its particles are objects of the class `MessageParticle`. A `MessageParticle` is simply a particle whose data field is an `Envelope`, which means that it can hold a `Message` in the same way that `Envelope` objects do.

Many methods of the `Particle` class are redefined in the `MessageParticle` class to cause a run-time error; for example, it is illegal to send an integer, floating, or complex number to the particle with the << operator. The conversion operators (conversion to type `int`, `double`, or `Complex`) return errors by default, but can be made legal by redefining the `asInt`, `asFloat`, or `asComplex` methods for a specific message type.

The principal operations on `MessageParticle` objects are << with an argument of type `Envelope`, to load a message into the particle, and `getMessage(Envelope&)`, to transfer message contents from the particle into a user-supplied message. The `getMessage` method removes the message contents from the particle[1]. In cases where the destructive behavior of `getMessage` cannot be tolerated, an alternative interface, `accessMessage(Envelope&)`, is provided. It does not remove the message contents from the particle. Promiscuous use of `accessMessage` in systems where large-sized messages may be present can cause the amount of virtual memory occupied to grow (though all message will be deleted eventually).

4.3.4 Use of messages in stars

Here are a couple of simple examples of stars that produce and consume messages. For more advanced samples, look in the Ptolemy distribution for stars that produce or consume messages. The image processing classes and stars, which are briefly described below in "Image particles" on page 4-40, provide a particularly rich set of examples. The matrix classes described on page 4-21 are also good examples. The matrix classes are recognized in the Ptolemy kernel, and supported by `pigi` and `ptlang`.

1. The reason for this "aggressive reclamation" policy (both here and in other places) is to minimize the number of no-longer-needed messages in the system and to prevent unnecessary clones from being generated by writableCopy() by eliminating references to Message objects as soon as possible.

```
defstar {
      name { PackInt }
      domain { SDF }
      desc { Accept integer inputs and produce IntVecData messages.}
      defstate {
            name { length }
            type { int }
            default { 10 }
            desc { number of values per message }
      }
      input {
            name { input }
            type { int }
      }
      output {
            name { output }
            type { message }
      }
      ccinclude { "Message.h", "IntVecData.h" }
      start {
            input.setSDFParams(int(length),int(length-1));
      }
      go {
            int l = length;
            IntVecData * pd = new IntVecData(l);
            // Fill in message. input%0 is newest, must reverse
            for (int i = 0; i < l; i++)
                  pd->data[l-i-1] = int(input%i);
            Envelope pkt(*pd);
            output%0 << pkt;
      }
}
```

Since this is an SDF star, it must produce and consume a constant number of tokens on each step, so the message length must be fixed (though it is controllable with a state). See "Setting SDF porthole parameters" on page 7-1 for an explanation of the `setSDFParams` method. Notice that the output porthole is declared to be of type `message`. Notice also the `ccinclude` statement; we must include the file `Message.h` in all message-manipulating stars, and we must also include the definition of the specific message type we wish to use.

The code itself is fairly straightforward—an `IntVecData` object is created with `new`, is filled in with data, and is put into an `Envelope` and sent. Resist the temptation to declare the `IntVecData` object as a local variable: it will not work. It must reside on the heap. Here is a star to do the inverse operation:

```
defstar {
      name { UnPackInt }
      domain { SDF }
      desc {
Accept IntVecData messages and produce integers. The first 'length'
values from each message are produced.
```

```
        }
        defstate {
              name { length }
              type { int }
              default { 10 }
              desc { number of values output per message }
        }
        input {
              name { input }
              type { message }
        }
        output {
              name { output }
              type { int }
        }
        ccinclude { "Message.h", "IntVecData.h" }
        start {
              output.setSDFParams(int(length),int(length-1));
        }
        go {
              Envelope pkt;
              (input%0).getMessage(pkt);
              if (!pkt.typeCheck("IntVecData")) {
                    Error::abortRun(*this,pkt.typeError("IntVecData"));
                    return;
              }
              const IntVecData * pd = (const IntVecData *)pkt.myData();
              if (pd.length() < int(length)) {
                    Error::abortRun(*this,
                          "Received message is too short");
                    return;
              }
              for (i = 0; i < int(length); i++) {
                    output%(int(length)-i-1) << pd->data[i];
              }
        }
  }
```

Because the domain is SDF, we must always produce the same number of outputs regardless of the size of the messages. The simple approach taken here is to require at least a certain amount of data or else to trigger an error and abort the run.

The operations here are to declare an envelope, get the data from the particle into the envelope with `getMessage`, check the type, and then access the contents. Notice the cast operation; this is needed because `myData` returns a const pointer to class `Message`. It is important that we converted the pointer to `const IntVecData *` and not `IntVecData*` because we have no right to modify the message through this pointer. Many C++ compilers will not warn by default about "casting away const"; we recommend turning on compiler warnings when compiling code that uses messages to avoid getting into trouble (for g++, say `-Wcast-qual`; for *cfront*-derived compilers, say `+w`).

If we wished to modify the message and then send the result as an output, we would call `writableCopy` instead of `myData`, modify the object, then send it on its way as in the previous star.

4.4 The Matrix Data Types

The primary support for matrix types in Ptolemy is the `PtMatrix` class. `PtMatrix` is derived from the `Message` class, and uses the various kernel support functions for working with the `Message` data type as described in Section 4.3 on page 4-14. This section discusses the `PtMatrix` class and how to write stars and programs using this class.

4.4.1 Design philosophy

The `PtMatrix` class implements two dimensional arrays. There are four key classes derived from `PtMatrix`: `ComplexMatrix`, `FixMatrix`, `FloatMatrix`, and `IntMatrix`. (Note that `FloatMatrix` is a matrix of C++ `doubles`.) A review of matrix classes implemented by other programmers revealed two main styles of implementation: a vector of vectors, or a simple array. In addition, there are two main formats of storing the entries: column-major ordering, where all the entries in the first column are stored before the entries of the second column, and row-major ordering, where the entries are stored starting with the first row. Column-major ordering is how Fortran stores arrays whereas row-major ordering is how C stores arrays.

The Ptolemy `PtMatrix` class stores data as a simple C array, and therefore uses row-major ordering. Row-major ordering also seems more natural for operations such as image and video processing, but it might make it more difficult to interface Ptolemy's `PtMatrix` class with Fortran library calls. The limits of interfacing Ptolemy's `PtMatrix` class with other software is discussed in Section 4.4.5 on page 4-33.

The design decision to store data entries in a C array rather than as an array of vector objects has a greater effect on performance than the decision whether to use row major or column major ordering. There are a couple of advantages to implementing a matrix class as an array of vector class objects: referencing an entry may be faster, and it is easier to do operations on a whole row or column of the matrix, depending on whether the format is an array of column vectors or an array of row vectors. An entry lookup in an array of row vectors requires two index lookups: one to find the desired row vector in the array and one to find the desired entry of that row. A linear array, in contrast, requires a multiplication to find the location of first element of the desired row and then an index lookup to find the column in that row. For example, `A[row][col]` is equivalent to looking up `&data + (row*numRows + col)` if the entries are stored in a C array `data[]`, whereas it is *(`&rowArray + row) + col` if looking up the entry in an array of vectors format.

Although the array of vectors format has faster lookups, it is also more expensive to create and delete the matrix. Each vector of the array must be created in the matrix constructor, and each vector must also be deleted by the matrix destructor. The array of vectors format also requires more memory to store the data and the extra array of vectors.

With the advantages and disadvantages of the two systems in mind, we chose to implement the `PtMatrix` class with the data stored in a standard C array. Ptolemy's environment is such that matrices are created and deleted constantly as needed by stars: this negates much of the speedup gained from faster lookups. Also, we felt it was important to keep the design of the class simple and the memory usage efficient because of Ptolemy's increasing size and complexity.

4.4.2 The PtMatrix class

The `PtMatrix` base class is derived from the `Message` class so that we can use Ptolemy's `Envelope` class and message-handling system. However, the `MessageParticle` class is not used by the `PtMatrix` class; instead, there are special `MatrixEnvParticle` classes defined to handle type checking between the various types of matrices. This allows the system to automatically detect when two stars with different matrix type inputs and outputs are incorrectly connected together.[1] Also, the `MatrixEnvParticle` class has some special functions not found in the standard `MessageParticle` class to allow easier handling of `PtMatrix` class messages. A discussion of how to pass `PtMatrix` class objects using the `MatrixEnvParticles` can be found in a following section.

As explained previously, there are currently four data-specific matrix classes: `ComplexMatrix`, `FixMatrix`, `FloatMatrix`, and `IntMatrix`. Each of these classes stores its entries in a standard C array named `data`, which is an array of data objects corresponding to the `PtMatrix` type: `Complex`, `Fix`, `double`, or `int`. These four matrix classes implement a common set of operators and functions; in addition, the `ComplexMatrix` class has a few special methods such as `conjugate()` and `hermitian()` and the `FixMatrix` class has a number of special constructors that allow the user to specify the precision of the entries in the matrix. Generally, all entries of a `FixMatrix` will have the same precision.

The matrix classes were designed to take full advantage of operator overloading in C++ so that operations on matrix objects can be written much like operations on scalar ones. For example, the two-operand multiply `operator *` has been defined so that if `A` and `B` are matrices, `A * B` will return a third matrix that is the matrix product of `A` and `B`.

4.4.3 Public functions and operators for the PtMatrix class

The functions and operators listed below are implemented by all matrix classes (`ComplexMatrix`, `FixMatrix`, `FloatMatrix`, and `IntMatrix`) unless otherwise noted. The symbols used are:

- `XXX` refers to one of the following: `Complex`, `Fix`, `Float`, or `Int`
- `xxx` refers to one of the following: `Complex`, `Fix`, `double`, or `int`

Functions and Operators to access entries of the Matrix:

`xxx& entry(int i)`

Example: `A.entry(i)`

Return the `i`th entry of the matrix when its data storage is considered to be a linear array. This is useful for quick operations on every entry of the `matrix` without regard for the specific (row,column) position of that entry. The total number of entries in the matrix is defined to be `numRows() * numCols()`, with indices ranging from 0 to `num-`

1. We recommend, however, that you do not adapt this method to your own types, but use the standard method of adding new message types described in Section 4.3. The method currently used for the matrix classes may not be supported in future releases.

`Rows() * numCols() - 1`. This function returns a reference to the actual entry in the matrix so that assignments can be made to that entry. In general, functions that wish to linearly reference each entry of a matrix `A` should use this function instead of the expression `A.data[i]` because classes which are derived from `PtMatrix` can then overload the `entry()` method and reuse the same functions.

`xxx* operator [] (int row)`

Example: `A[row][column]`

Return a pointer to the start of the row in the matrix's data storage. (This operation is different to matrix classes defined as arrays of vectors, in which the `[]` operator returns the vector representing the desired row.) This operator is generally not used alone but with the `[]` operator defined on C arrays, so that `A[i][j]` will give you the entry of the matrix in the i^{th} row and j^{th} column of the data storage. The range of rows is from 0 to `numRows()-1` and the range of columns is from 0 to `numCols()-1`.

Constructors:

`XXXMatrix()`

Example: `IntMatrix A;`

Create an uninitialized matrix. The number of rows and columns are set to zero and no memory is allocated for the storage of data.

`XXXMatrix(int numRow, int numCol)`

Example: `FloatMatrix A(3,2);`

Create a matrix with dimensions `numRow` by `numCol`. Memory is allocated for the data storage but the entries are uninitialized.

`XXXMatrix(int numRow, int numCol, PortHole& p)`

Example: `ComplexMatrix(3,3,myPortHole)`

Create a matrix of the given dimensions and initialize the entries by assigning to them values taken from the porthole `myPortHole`. The entries are assigned in a rasterized sequence so that the value of the first particle removed from the porthole is assigned to entry (0,0), the second particle's value to entry (0,1), etc. It is assumed that the porthole has enough particles in its buffer to fill all the entries of the new matrix.

`XXXMatrix(int numRow, int numCol, XXXArrayState& dataArray)`

Example: `IntMatrix A(2,2,myIntArrayState);`

Create a matrix with the given dimensions and initialize the entries to the values in the given `ArrayState`. The values of the `ArrayState` fill the matrix in rasterized sequence so that entry (0,0) of the matrix is the first entry of the `ArrayState`, entry (0,1) of the matrix is the second, etc. An error is generated if the `ArrayState` does not have enough values to initialize the whole matrix.

`XXXMatrix(const XXXMatrix& src)`

Example: `FixMatrix A(B);`
This is the copy constructor. A new matrix is formed with the same dimensions as the source matrix and the data values are copied from the source.

`XXXMatrix(const XXXMatrix& src, int startRow, int startCol, int numRow, int numCol)`
Example: `IntMatrix A(B,2,2,3,3);`
This special "submatrix" constructor creates a new matrix whose values come from a submatrix of the source. The arguments `startRow` and `startCols` specify the starting row and column of the source matrix. The values `numRow` and `numCol` specify the dimensions of the new matrix. The sum `startRow + numRow` must not be greater than the maximum number of rows in the source matrix; similarly, `startCol + numCol` must not be greater than the maximum number of columns in the source. For example, if `B` is a matrix with dimension (4,4), then `A(B,1,1,2,2)` would create a new matrix `A` that is a (2,2) matrix with data values from the center quadrant of matrix `B`, so that `A[0][0] == B[1][1]`, `A[0][1] == B[1][2]`, `A[1][0] == B[2][1]`, and `A[1][1] == B[2][2]`.

The following are special constructors for the `FixMatrix` class that allow the programmer to specify the precision of the entries of the `FixMatrix`.

`FixMatrix(int numRow, int numCol, int length, int intBits)`
Example: `FixMatrix A(2,2,14,4);`
Create a `FixMatrix` with the given dimensions such that each entry is a fixed-point number with precision as given by the `length` and `intBits` inputs.

`FixMatrix(int numRow, int numCol, int length, int intBits, PortHole& myPortHole)`
Example: `FixMatrix A(2,2,14,4);`
Create a `FixMatrix` with the given dimensions such that each entry is a fixed-point number with precision as given by the `length` and `intBits` inputs and initialized with the values that are read from the particles contained in the porthole `myPortHole`.

`FixMatrix(int numRow, int numCol, int length, int intBits, FixArrayState& dataArray)`
Example: `FixMatrix A(2,2,14,4);`
Create a `FixMatrix` with the given dimensions such that each entry is a fixed-point number with precision as given by the `length` and `intBits` inputs and initialized with the values in the given `FixArrayState`.

There are also special copy constructors for the `FixMatrix` class that allow the programmer to specify the precision of the entries of the `FixMatrix` as they are copied from the sources. These copy constructors are usually used for easy conversion between the other matrix types.

The last argument specifies the type of masking function (truncate, rounding, etc.) to be used when doing the conversion.

`FixMatrix(const XXXMatrix& src, int length, int intBits, int round)`

Example: `FixMatrix A(CxMatrix,4,14,TRUE);`

Create a `FixMatrix` with the given dimensions such that each entry is a fixed-point number with precision as given by the `length` and `intBits` arguments. Each entry of the new matrix is copied from the corresponding entry of the src matrix and converted as specified by the `round` argument.

Comparison operators:

`int operator == (const XXXMatrix& src)`

Example: `if(A == B) then` ...

Return `TRUE` if the two matrices have the same dimensions and every entry in `A` is equal to the corresponding entry in `B`. Return `FALSE` otherwise.

`int operator != (const XXXMatrix& src)`

Example: `if(A != B) then` ...

Return `TRUE` if the two matrices have different dimensions or if any entry of `A` differs from the corresponding entry in `B`. Return `FALSE` otherwise.

Conversion operators:

Each matrix class has a conversion operator so that the programmer can explicitly cast one type of matrix to another (this casting is done by copying). It would have been possible to make conversions occur automatically when needed, but because these conversions can be quite expensive for large matrices, and because unexpected results might occur if the user did not intend for a conversion to occur, we chose to require that these conversions be used explicitly.

`operator XXXMatrix () const`

Example: `FloatMatrix C = A * (FloatMatrix)B;`

Convert a matrix of one type into another. These conversions allow the various arithmetic operators, such as `*` and `+`, to be used on matrices of different type. For example, if `A` in the example above is a (3,3) `FloatMatrix` and `B` is a (3,2) `IntMatrix`, then `C` is a `FloatMatrix` with dimensions (3,2).

Destructive replacement operators:

These operators are member functions that modify the current value of the object. In the following examples, `A` is usually the lvalue (`*this`). All operators return `*this`:

`XXXMatrix& operator = (const XXXMatrix& src)`

Example: `A = B;`

This is the assignment operator: make `A` into a matrix that is a copy of `B`. If `A` already has allocated data storage, then the size of this data storage is compared to the size of `B`. If they are equal, then the dimensions of `A` are simply set to those of `B` and the entries copied. If they are not equal, the data storage is freed and reallocated before copying.

`XXXMatrix& operator = (xxx value)`
Example: `A = value;`
Assign each entry of `A` to have the given `value`. Memory management is handled as in the previous operator.
Note: this operator is targeted for deletion. Do not use it.

`XXXMatrix& operator += (const XXXMatrix& src)`
Example: `A += B;`
Perform the operation `A.entry(i) += B.entry(i)` for each entry in `A`. `A` and `B` must have the same dimensions.

`XXXMatrix& operator += (xxx value)`
Example: `A += value;`
Add the scalar `value` to each entry in the matrix.

`XXXMatrix& operator -= (const XXXMatrix& src)`
Example: `A -= B;`
Perform the operation `A.entry(i) -= B.entry(i)` for each entry in `A`. `A` and `B` must have the same dimensions.

`XXXMatrix& operator -= (xxx value)`
Example: `A -= value;`
Subtract the scalar `value` from each entry in the matrix.

`XXXMatrix& operator *= (const XXXMatrix& src)`
Example: `A *= B;`
Perform the operation `A.entry(i) *= B.entry(i)` for each entry in `A`. `A` and `B` must have the same dimensions. Note: this is an elementwise operation and is *not* equivalent to A = A * B.

`XXXMatrix& operator *= (xxx value)`
Example: `A *= value;`
Multiply each entry of the matrix by the scalar `value`.

`XXXMatrix& operator /= (const XXXMatrix& src)`
Example: `A /= B;`
Perform the operation `A.entry(i) /= B.entry(i)` for each entry in `A`. `A` and `B` must have the same dimensions.

`XXXMatrix& operator /= (xxx value)`
`Example: A /= value`
Divide each entry of the matrix by the scalar `value`. The scalar value must be non-zero.

`XXXMatrix& operator identity()`
Example: `A.identity();`
Change `A` to be an identity matrix so that each entry on the diagonal is 1 and all off-diagonal entries are 0.

Non-destructive operators (these return a new matrix):

`XXXMatrix operator - ()`
Example: `B = -A;`
Return a new matrix such that each element is the negative of the element of the source.

`XXXMatrix operator ~ ()`
Example: `B = ~A;`
Return a new matrix that is the transpose of the source.

`XXXMatrix operator ! ()`
Example: `B = !A;`
Return a new matrix which is the inverse of the source.

`XXXMatrix operator ^ (int exponent)`
Example: `B = A^2;`
Return a new matrix which is the source matrix to the given `exponent` power. The `exponent` can be negative, in which case the `exponent` is first treated as a positive number and the final result is then inverted. So `A^2 == A*A` and `A^(-3) == !(A*A*A).`

`XXXMatrix transpose()`
Example: `B = A.transpose();`
This is the same as the `~ operator` but called by a function name instead of as an operator.

`XXXMatrix inverse()`
Example: `B = A.inverse();`
This is the same as the `! operator` but called by a function name instead of as an operator.

`ComplexMatrix conjugate()`
Example: `ComplexMatrix B = A.conjugate();`
Return a new matrix such that each element is the complex conjugate of the source. This function is defined for the `ComplexMatrix` class only.

`ComplexMatrix hermitian()`
Example: `ComplexMatrix B = A.hermitian();`
Return a new matrix which is the Hermitian Transpose (conjugate transpose) of the source. This function is defined for the `ComplexMatrix` class only.

Non-member binary operators:

`XXXMatrix operator + (const XXXMatrix& left, const XXXMatrix& right)`

Example: `A = B + C;`

Return a new matrix which is the sum of the first two. The `left` and `right` source matrices must have the same dimensions.

`XXXMatrix operator + (const xxx& scalar, const XXXMatrix& matrix)`

Example: `A = 5 + B;`

Return a new matrix that has entries of the `source` matrix added to a `scalar` value.

`XXXMatrix operator + (const XXXMatrix& matrix, const xxx& scalar)`

Example: `A = B + 5;`

Return a new matrix that has entries of the source matrix added to a scalar value. (This is the same as the previous operator but with the `scalar` on the right.)

`XXXMatrix operator - (const XXXMatrix& left, const XXXMatrix& right)`

Example: `A = B - C;`

Return a new matrix which is the difference of the first two. The `left` and `right` source matrices must have the same dimensions.

`XXXMatrix operator - (const xxx& scalar, const XXXMatrix& matrix)`

Example: `A = 5 - B;`

Return a new matrix that has the negative of the entries of the source `matrix` added to a `scalar` value.

`XXXMatrix operator - (const XXXMatrix& matrix, const xxx& scalar)`

Example: `A = B - 5;`

Return a new matrix such that each entry is the corresponding entry of the source `matrix` minus the `scalar` value.

`XXXMatrix operator * (const XXXMatrix& left, const XXXMatrix& right)`

Example: `A = B * C;`

Return a new matrix which is the matrix product of the first two. The `left` and `right` source matrices must have compatible dimensions (i.e. `A.numCols() == B.numRows()`.

`XXXMatrix operator * (const xxx& scalar, const XXXMatrix& matrix)`

Example: `A = 5 * B;`

Return a new matrix that has entries of the source `matrix` multiplied by a `scalar` value.

`XXXMatrix operator * (const XXXMatrix& matrix, const xxx& scalar)`

Example: `A = B * 5;`

Return a new matrix that has entries of the source matrix multiplied by a scalar value. (This is the same as the previous operator but with the `scalar` on the right.)

Miscellaneous functions:

`int numRows()`

Return the number of rows in the matrix.

`int numCols()`

Return the number of columns in the matrix.

`Message* clone()`

Example: `IntMatrix *B = A.clone();`

Return a copy of `*this`.

`StringList print()`

Example: `A.print()`

Return a formatted `StringList` that can be printed to display the contents of the matrix in a reasonable format.

`XXXMatrix& multiply (const XXXMatrix& left, const XXXMatrix& right, XXXMatrix& result)`

Example: `multiply(A,B,C);`

This is a faster 3 operand form of matrix multiply such that the result matrix is passed as an argument so that we avoid the extra copy step that is involved when we write `C = A * B`.

`const char* dataType()`

Example: `A.dataType()`

Return a string that specifies the name of the type of matrix. The strings are "`ComplexMatrix`", "`FixMatrix`", "`FloatMatrix`", and "`IntMatrix`".

`int isA(const char* type)`

Example: `if(A.isA("FixMatrix")) then ...`

Return `TRUE` if the argument string matches the type string of the matrix.

4.4.4 Writing stars and programs using the PtMatrix class

This section describes how to use the matrix data classes when writing stars. Some examples will be given here but the programmer should refer to the stars in `$PTOLEMY/src/domains/sdf/matrix/stars/*.pl` and `$PTOLEMY/src/domains/sdf/image/`

`stars/*.pl` for more examples

Memory management

The most important thing to understand about the use of matrix data classes in the Ptolemy environment is that stars that intend to output the matrix in a particle should allocate memory for the matrix *but never delete that matrix*. Memory reclamation is done automatically by the reference-counting mechanism of the `Message` class. Strange errors will occur if the star deletes the matrix before it is used by another star later in the execution sequence.

Naming conventions

Stars that implement general-purpose matrix operations usually have names with the `_M` suffix to distinguish them from stars that operate on scalar particles. For example, the `SDFGain_M` star multiplies an input matrix by a scalar value and outputs the resulting matrix. This is in contrast to `SDFGain`, which multiplies an input value held in a `FloatParticle` by a double and puts that result in an output `FloatParticle`.

Include files

For a star to use the `PtMatrix` classes, it must include the file `Matrix.h` in either its `.h` or `.cc` file. If the star has a matrix data member, then the declaration

```
hinclude { "Matrix.h" }
```

needs to be in the `Star` definition. Otherwise, the declaration

```
ccinclude { "Matrix.h" }
```

is sufficient.

To declare an input porthole that accepts matrices, the following syntax is used:

```
input {
      name { inputPortHole }
      type { FLOAT_MATRIX_ENV }
}
```

The syntax is the same for output portholes. The type field can be `COMPLEX_MATRIX_ENV`, `FLOAT_MATRIX_ENV`, `FIX_MATRIX_ENV`, or `INT_MATRIX_ENV`. The icons created by Ptolemy will have terminals that are thicker and that have larger arrow points than the terminals for scalar particle types. The colors of the terminals follow the pattern of colors for scalar data types (e.g., blue represents `Float` and `FloatMatrix`).

Input portholes

The syntax to extract a matrix from the input porthole is:

```
Envelope inPkt;
(inputPortHole%0).getMessage(inPkt);
const FloatMatrix& inputMatrix =
      *(const FloatMatrix *)inPkt.myData();
```

The first line declares an `Envelope`, which is used to access the matrix. Details of the `Envelope` class are given in “Use of the Envelope class” on page 4-17. The second line fills the envelope with the input matrix. Note that, because of the reference-counting mechanism, this line does not make a copy of the matrix. The last two lines extract a reference to the matrix from the envelope. It is up to the programmer to make sure that the cast agrees with the definition of the input port.

Because multiple envelopes might reference the same matrix, a star is generally not permitted to modify the matrix held by the `Envelope`. Thus, the function `myData()` returns a `const Message *`. We cast that to be a `const FloatMatrix *` and then de-reference it and assign the value to `inputMatrix`. It is generally better to handle matrices by reference instead of by pointer because it is clearer to write “`A + B`” rather than “`*A + *B`” when working with matrix operations. Stars that wish to modify an input matrix should access it using the `writableCopy` method, as explained in “Use of the Envelope class” on page 4-17.

Allowing delays on inputs

The cast to `(const FloatMatrix *)` above is not always safe. Even if the source star is known to provide matrices of the appropriate type, a delay on the arc connecting the two stars can cause problems. In particular, delays in dataflow domains are implemented as initial particles on the arcs. These initial particles are given the value “zero” as defined by the type of particle. For `Message` particles, a “zero” is an uninitialized `Message` particle containing a “dummy” data value. This dummy `Message` will be returned by the `myData` method in the third line of the above code fragment. The dummy message is not a `FloatMatrix`, rendering the above cast invalid. A star that expects matrix inputs must have code to handle empty particles. An example is:

```
if(inPkt.empty()) {
      FloatMatrix& result = *(new FloatMatrix(int(numRows),
                                        int(numCols)));
      result = 0.0;
      output%0 << result;
}
```

There are many ways that an empty input can be interpreted by a star that operates on matrices. For example, a star multiplying two matrices can simply output a zero matrix if either input is empty. A star adding two matrices can output whichever input is not empty. Note above that we create an output matrix that has the dimensions as set by the state parameters of the star so that any star that uses this output will have valid data.

A possible alternative to outputting a zero matrix is to simply pass that empty `MessageParticle` along. This approach, however, can lead to counterintuitive results. Suppose that empty message reaches a display star like `TkText`, which will attempt to call the `print()` method of the object. An empty message has a `print()` method that results in a message like

<*type*>: no print method

This is likely to prove extremely confusing to users, so we strongly recommend that each matrix star handle the empty input in a reasonable way, and produce a non-empty output.

Matrix outputs

To put a matrix into an output porthole, the syntax is:

```
FloatMatrix& outMatrix =*(new FloatMatrix(someRow,someCol));
```

```
                // ... do some operations on the outMatrix
        outputPortHole%0 << outMatrix;
```

The last line is similar to outputting a scalar value. This is because we have overloaded the << operator for MatrixEnvParticles to support PtMatrix class inputs. The standard use of the MessageParticle class requires you to put your message into an envelope first and then use << on the envelope (see “Use of the Envelope class” on page 4-17), but we have specialized this so that the extra operation of creating an envelope first is not explicit.

Here is an example of a complete star definition that inputs and outputs matrices:

```
defstar {
      name { Mpy_M }
      domain { SDF }
      desc {
Does a matrix multiplication of two input Float matrices A and B to
produce matrix C.
      Matrix A has dimensions (numRows,X).
      Matrix B has dimensions (X,numCols).
      Matrix C has dimensions (numRows,numCols).
The user need only specify numRows and numCols. An error will be
generated automatically if the number of columns in A does not match
the number of columns in B.
      }
      input {
            name { Ainput }
            type { FLOAT_MATRIX_ENV }
      }
      input {
            name { Binput }
            type { FLOAT_MATRIX_ENV }
      }
      output {
            name { output }
            type { FLOAT_MATRIX_ENV }
      }
      defstate {
            name { numRows }
            type { int }
            default { 2 }
            desc { The number of rows in Matrix A and Matrix C.}
      }
      defstate {
            name { numCols }
            type { int }
            default { 2 }
            desc { The number of columns in Matrix B and Matrix C}
      }
      ccinclude { "Matrix.h" }
      go {
            // get inputs
            Envelope Apkt;
            (Ainput%0).getMessage(Apkt);
            const FloatMatrix& Amatrix =
```

```
                *(const FloatMatrix *)Apkt.myData();

        Envelope Bpkt;
        (Binput%0).getMessage(Bpkt);
        const FloatMatrix& Bmatrix =
                *(const FloatMatrix *)Bpkt.myData();

        // check for "null" matrix inputs, which could be
        // caused by delays on the input line
        if(Apkt.empty() || Bpkt.empty()) {
                // if either input is empty, return a zero
                // matrix with the state dimensions
                FloatMatrix& result =
                        *(new FloatMatrix(int(numRows),
                                        int(numCols)));
                result = 0.0;
                output%0 << result;
        }
        else {
                // Amatrix and Bmatrix are both valid
                if((Amatrix.numRows() != int(numRows)) ||
                        (Bmatrix.numCols() != int(numCols))) {
                        Error::abortRun(*this,
                        "Dimension size of FloatMatrix inputs do ",
                        "not match the given state parameters.");
                        return;
                }
                // do matrix multiplication
                FloatMatrix& result =
                        *(new FloatMatrix(int(numRows),
                                        int(numCols)));
                // we could write
                //      result = Amatrix * Bmatrix;
                // but the following is faster
                multiply(Amatrix,Bmatrix,result);

                output%0 << result;
        }
    }
}
```

4.4.5 Future extensions

After reviewing the libraries of numerical analysis software that is freely available on the Internet, it is clear that it would be beneficial to extend the `PtMatrix` class by adding those well-tested libraries as callable functions. Unfortunately, many of those libraries are currently only available in Fortran, and there are some incompatibilities with Fortran's column major ordering and C's row major ordering. Those problems would still exist even if the Fortran code was converted to C. There are a few groups which are currently working on C++ ports of the numerical analysis libraries. One notable group is the Lapack++[1] project which is

1. **LAPACK++: A Design Overview of Object-Oriented Extensions for High Performance Linear Algebra**, by Jack J. Dongarra, Roldan Pozo, and David W. Walker, available on *netlib*.

developing a flexible matrix class of their own, besides porting the Fortran algorithms of Lapack into C++. This might possibly be incorporated in a future release.

4.5 The File and String Types

There are two experimental types in Ptolemy that support non-numeric computation. These types represent the beginnings of an effort to extend Ptolemy's dataflow model to "non-dataflow" problems such as scheduling and design flow. Their interfaces are still being developed, so should be expected to change in future releases. We would welcome suggestions on how to improve the interface and functionality of these two types.

4.5.1 The File type

The file type is implemented by the classes `FileMessage` and `FileParticle`, which are derived from `Message` and `Particle`. It uses the reference-counting mechanism of the `Message` and `Envelope` classes to ensure that files are not deleted until no longer needed. Although we created a new particle type to allow these types to appear in the `pigi` graphical interface, we recommend that you use the `Message` interface described in Section 4.3 for your own types.

The `File` type adds the following functions to `Message`:

Constructors

`FileMessage()`
: Create a new file message with a unique filename. By default, the file will be deleted when no file messages reference it.

`FileMessage(const char* name)`
: Create a new file message with the given filename. By default, the file will not be deleted when no file messages reference it.

`FileMessage(const FileMessage& src)`
: Create a new file message containing the same filename as the given file message. By default, the file will not be deleted when no file messages reference it.

Operations

`const char* fileName()`
: Return the file name contained in this message.

`StringList print()`
: Return the file name contained in this message in a `StringList` object.

`const char* fileName()`
: Return the file name contained in this message.

`void setTransient(int transient)`
: Set the status of the file. If transient is `TRUE`, the file will be deleted

when no file messages reference it; if `FALSE`, then it will not be deleted.

4.5.2 The String type

The string type is implemented by the classes `StringMessage` and `StringParticle`, which are derived from `Message` and `Particle`. It contains an `InfString` object—`InfString` is a version of `StringList` that allows limited modification, and is used to interface C++ to Tcl. Again, It uses the reference-counting mechanism of the `Message` and `Envelope` classes to ensure that strings are not deleted until no longer needed. `StringMessage` is currently very simple—it adds the following functions to `Message`:

Constructors

`StringMessage()`
Create a new string message an empty string.

`StringMessage(const char* name)`
Create a new string message with a copy of the given string. The given string can be deleted, since the new message does not reference it.

`StringMessage(const StringMessage& src)`
Create a new string message containing the same string as the given string message. Again, the string is copied.

Operations

`StringList print()`
Return the string contained in this message in a `StringList` object.

4.6 Writing Stars That Manipulate Any Particle Type

Ptolemy allows stars to declare that inputs and outputs are of type `ANYTYPE`. A star may need to do this, for example, if it simply copies its inputs without regard to type, as in the case of a `Fork` star, or if it calls a generic function that is overloaded by every data type, such as sink stars which call the print method of the type.

The following is an example of a star that operates on `ANYTYPE` particles:

```
defstar {
      name {Fork}
      domain {SDF}
      desc { Copy input particles to each output. }
      input {
            name{input}
            type{ANYTYPE}
      }
      outmulti {
            name{output}
            type{= input}
      }
      go {
```

```
                MPHIter nextp(output);
                PortHole* p;
                while ((p = nextp++) != 0)
                      (*p)%0 = input%0;
          }
    }
```

Notice how in the definition of the output type, the star simply says that its output type will be the same as the input type. ptlang translates this definition into an `ANYTYPE` output porthole and a statement in the star constructor that reads

```
output.inheritTypeFrom(input);
```

as you can see by examining the `.cc` file generated for `SDFFork`.

During galaxy setup, the Ptolemy kernel assigns actual types to `ANYTYPE` portholes, making use of the types of connected portholes and inheritTypeFrom connections. For example, if a Fork's input is connected to an output porthole of type `INT`, the Fork's input becomes type `INT`, and then so do its output(s) thanks to the inheritTypeFrom connection. At runtime there is no such thing as an `ANYTYPE` porthole; every porthole has been resolved to some specific data type, which can be obtained from the porthole using the `resolvedType()` method. (However, this mechanism does not distinguish among the various subclasses of `Message`, so if you are using `Message` particles you still need to check the actual type of each `Message` received.)

Porthole type assignment is really a fairly complex and subtle algorithm, which is discussed further in the Ptolemy Kernel Manual. The important properties for a star writer to know are these:

- If an input port has a specific declared type, it is guaranteed to receive particles of that type. For reasons mentioned in "Reading inputs and writing outputs" on page 2-17, it is safest to explicitly cast input particles to the desired type, as in

  ```
  go {
        double value = double(in%0);
        ...
  }
  ```

  but this is not strictly necessary in the current system.

- In simulation domains, an output port is NOT guaranteed to transmit particles of its declared type; the actual resolved type of the porthole will be determined by the connected input porthole. Therefore, you should always allow for type conversion of the value computed by the star into the actual type of the output particle. This happens implicitly when you write something like

  ```
  out%0 << t;
  ```

  because this expands into a call of the particle's virtual method for loading a value of the given type. But assuming that you know the exact type of particle in the porthole --- say by writing something like `(FloatParticle&) (out%0)` --- is very unsafe.

- In code generation domains, it is usually critical that the output porthole's actual type be what the star writer expected. Most codegen domains therefore splice type conversion stars into the schematic when input and output ports of different declared types are connected. In this way, both connected stars will see the data type they expect, and the necessary type conversion is handled transparently.

- The component portholes of a multiporthole are type-resolved separately. Thus, if an input multiporthole is declared `ANYTYPE`, its component portholes might have different types at runtime. (This was not true in Ptolemy versions preceding 0.7.) The component portholes of an output multiporthole can have different resolved types in any case, because they might be connected to inputs of different types.
- It is rarely a good idea to declare a pure `ANYTYPE` output porthole; rather, its type should be equated to some input porthole using the ptlang `= port` notation or an explicit inheritTypeFrom call. This ensures that the type resolution algorithm can succeed. A "pure `ANYTYPE`" output will work only if connected to an input of determinable type; if it's connected to an `ANYTYPE` input, the kernel will be unable to resolve a type for the connection. By providing an `= type` declaration, you allow the kernel to choose an appropriate particle type for an `ANYTYPE`-to-`ANYTYPE` connection.

4.7 Unsupported Types

There are a number of data types in Ptolemy that we recommend not be used by external developers because they are either insufficiently mature or likely to change. This section briefly describes those classes.

4.7.1 Sub-matrices

The Ptolemy kernel contains a set of matrices to support efficient computation with sub-matrices. These classes were developed specifically for the experimental multidimensional SDF (MDSDF) domain and will probably be implemented differently in a future release.

There are four sub-matrix classes, one for each concrete matrix class: `ComplexSubMatrix`, `FixSubMatrix`, `FloatSubMatrix`, and `IntSubMatrix`, each of which inherits from the corresponding `PtMatrix` class. A sub-matrix contains a reference to a "parent" matrix of the same type, and modifies its internal data pointers and matrix size parameters to reference a rectangular region of the parent's data. The constructors for the submatrix classes have arguments that specify the region of the parent matrix referenced by the sub-matrix.

As for matrices, the description of sub-matrices uses the convention that `XXX` means `Complex`, `Fix`, `Float`, or `Int`, and `xxx` means `Complex`, `Fix`, `double`, or `int`.

The submatrix constructors are:

`XXXSubMatrix()`
: Create an uninitialized matrix.

`XXXSubMatrix(int numRow, int numCol)`
: Create a regular matrix with dimensions `numRow` by `numCol`; return a new submatrix with this matrix as its parent. Memory is allocated for the data storage but the entries are uninitialized.

`XXXSubMatrix(XXXSubMatrix& src, int sRow, int sCol, int nRow,` int nCol)

Create a sub-matrix of the given dimensions and initialize it to reference the region of the parent matrix starting at (sRow, sCol) and of size (nRow, nCol). The parent matrix is the same as the parent matrix of src. The given dimensions must fit into the parent matrix, or an error will be flagged. Unlike the “sub-matrix” constructors in the regular matrix classes, this constructor does not copy matrix data.

XXXSubMatrix(const XXXSubMatrix& src)

Make a duplicate of the src sub-matrix. The parent of the new matrix is the same as the parent of src.

Submatrices support all operations supported by the regular matrix classes. Because the matrix classes uniformly use only the entry() and operator [] member functions to access the data, the sub-matrix classes need only to override these functions, and all matrix operations become available on sub-matrices.

xxx& entry(int i)

Return the i^{th} entry of the sub-matrix when its data storage is considered to be a linear array.

xxx* operator [] (int row)

Return a pointer to the start of the row of the sub-matrix’s data storage.

Using sub-matrices in stars

Sub-matrices are not currently useful in general-purpose dataflow stars. Rather, they were developed to provide an efficient means of referencing portions of a single larger matrix in the multi-dimensional synchronous dataflow (MDSDF) domain. We give here a summary. For more details, see [Che94] and the MDSDF sources in $PTOLEMY/src/domains/mdsdf/kernel and $PTOLEMY/src/domains/mdsdf/stars.

Unlike other domains, the MDSDF kernel does not transfer particles through FIFO buffers. Instead, each geodesic keeps a single copy of a “parent” matrix, that represents the “current” two-dimensional datablock. Each time a star fires, it obtains a sub-matrix that references this parent matrix with the getOutput() function of the MDSDF input port class. For example, a star might contain:

```
FloatSubMatrix* data = (FloatSubMatrix*)(input.getInput());
```

Note that this is not really getting a matrix, but a sub-matrix that references a region of the current data matrix. The size of the sub-matrix has been set by the star in its initialization code by calling the setMDSDFParams() function of the port.

To write data to the output matrix, the star gets a sub-matrix which references a region of the current output matrix and writes to it with a matrix operator. For example,

```
FloatSubMatrix* result = (FloatSubMatrix*)(output.getOutput());
result = -data;
```

Because the sub-matrices are only references to the current matrix on each arc they must be deleted after use:

```
delete &input;
delete &result;
```

Here is a simplified example of a complete MDSDF star:

```
defstar {
      name { Add }
      domain { MDSDF }
      desc {
            Matrix addition of two input matrices A and B to
            produce matrix C. All matrices must have the same
            dimensions.
      }
      version { %W% %G% }
      author { Mike J. Chen }
      location  { MDSDF library }
      input {
            name { Ainput }
            type { FLOAT_MATRIX }
      }
      input {
            name { Binput }
            type { FLOAT_MATRIX }
      }
      output {
            name { output }
            type { FLOAT_MATRIX }
      }
      defstate {
            name { numRows }
            type { int }
            default { 2 }
            desc { The number of rows in the input/output matrices. }
      }
      defstate {
            name { numCols }
            type { int }
            default { 2 }
            desc { The number of columns in the input/output
                        matrices. }
      }
      ccinclude { "SubMatrix.h" }
      setup {
          Ainput.setMDSDFParams(int(numRows), int(numCols));
          Binput.setMDSDFParams(int(numRows), int(numCols));
          output.setMDSDFParams(int(numRows), int(numCols));
      }
      go {
      // get a SubMatrix from the buffer
            FloatSubMatrix& input1
```

```
                        = *(FloatSubMatrix*)(Ainput.getInput());
                FloatSubMatrix& input2
                        = *(FloatSubMatrix*)(Binput.getInput());
                FloatSubMatrix& result
                        = *(FloatSubMatrix*)(output.getOutput());

        // compute product, putting result into output

                result = input1 + input2;

                delete &input1;
                delete &input2;
                delete &result;
            }
        }
```

The sub-matrix "particles"

The `ptlang` type of submatrices is `FLOAT_MATRIX`, `INT_MATRIX`, and so on. (This is not documented in the *User's Manual* and is likely to change in a future release.) Each of these ptlang types is implemented by a sub-class of Particle: `IntMatrixParticle`, `FloatMatrixParticle`, `FixMatrixParticle` and `ComplexMatrixParticle`. These particle classes exist only for setting up the portholes and performing type-checking—they are never created or passed around during a simulation. Instead, sub-matrices are created and destroyed by the MDSDF kernel and stars as described above.

4.7.2 Image particles

A set of experimental image data types, designed to make it convenient to manipulate images and video sequences in Ptolemy, were defined by Paul Haskell. They are based on Ptolemy's built-in `Message` type, described above. A library of stars that uses these image data types can be found in the image library of the DE domain.

This set of classes is being replaced by the `PtMatrix` classes, and the SDF image classes now all use `PtMatrix`. We give here a brief introduction to the image data types used in the DE domain, although new work should consider using `PtMatrix` classes instead. Class definitions can be found in `$PTOLEMY/src/domains/de/kernel`.

The base class of all the image classes is called `BaseImage`. It has some generic methods and members for manipulating images. Most of the methods are redefined in the derived classes. The `fragment` method partitions an image into many smaller images, which together represent the same picture as the original. The `assemble` method combines many small images which make up a single picture into a single image that contains the picture. The `fragment` method works recursively, so an image that has been produced by a previous `fragment` call can be further fragmented. Assembly always produces a full-sized image from fragments, however small.

Use of the `size`, `fullSize`, and `startPos` members varies within each subclass. Typically the `size` variable holds the number of pixels that an object is storing. If an object is not produced by `fragment()`, then (`size == fullSize`). If the object is produced by a `fragment()` call, `size` may be less than or equal to `fullSize`. An objects's `fullSize` may be bigger or smaller than `width*height`. It would be bigger, for example, in `DCTIm-`

age, where the amount of allocated storage must be rounded up to be a multiple of the blocksize. It would be smaller, for example, for an object that contains run-length coded video.

The frameId variable is used during assembly. Fragments with the same frameId's are assembled into the same image. So, it is important that different frames from the same source have different frameIds.

The comparison functions {==, !=, <, >, etc.} compare two objects' frameId's. They can be used to resequence images or to sort image fragments.

The copy constructor and clone methods have an optional integer argument. If a nonzero argument is provided, then all state values of the copied object are copied to the created object, but none of the image data is copied. If no argument or a zero argument is provided, then the image data is copied as well. Classes derived from BaseImage should maintain this policy.

The GrayImage class, derived from BaseImage, is used to represent gray-scale images. The DCTImage class is used to represent images or image fragments that have been encoded using the discrete-cosine transform. The MVImage class is a bit more specialized; it stores a frame's worth of motion vectors.

4.7.3 "First-class" types

All of the types built-in to the Ptolemy kernel are "first-class" in the sense that they are understood by pigi and ptlang. We recommend that users create their own types using the mechanism described in "Defining New Data Types" on page 4-14. This approach has the disadvantage that all user-defined types are seen by pigi and ptlang as being of type "message." If this is not acceptable, then it is possible to create your own first-class types by subclassing Particle and adding the new types to VEM. The following instructions briefly describes this process. We stress, however, that this method is not officially supported and that types created this way will probably have to be reworked in a future release of Ptolemy. You will need to use some other color—say fileColor—as a sample to follow when modifying the various source files.

- Sub-class Particle and Message. Use the classes in $PTOLEMY/src/kernel/FileMessage.h/cc and $PTOLEMY/src/kernel/FileParticle.{h,cc} as examples. You will need to create a static instance of your Particle and static Plasma and PlasmaGate instances to hold your particles, as demonstrated by FileParticle.
- Modify $PTOLEMY/src/pigilib/mkTerm.c. There are three switch statements where you will need to insert a new case.
- In the directory $PTOLEMY/lib/colors/ptolemy, edit bw.pat and colors.pat to add the new color. The color is in RBG format, with 1000 being full-scale.
- Run the Octtools installColors program. It will ask you a series of mysterious and strangely beautiful questions. To start with, use the defaults, except for "Output display type", where you answer GENERIC-COLOR. Run the same program again with the following output display types: GENERIC-BW, Postscript-Color, and Postscript-BW.
- To support monochrome screens (when pigi is started with the -bw option), repeat

the above, but specify `$PTOLEMY/lib/colors/ptolemy/bw.pat` as the pattern file, `$PTOLEMY/lib/bw_patterns` as the directory in which to install, `GENERIC-COLOR` as the display device, and answer `YES` to the question about color output device.

- After rebuilding `pigilib` and restarting, create an icon for a star that has your new type as an input or output. The terminal should be of the new color.