

Playa User's Manual

A motley crew

February 10, 2011

Contents

1 Overview	2
1.1 Principal user-level objects	2
1.1.1 Handles, deep and shallow copies	4
1.1.2 Example: A conjugate gradient solver	5
1.1.3 Example: Inverse power iteration	6
2 Vectors	7
2.1 Creation of vectors	7
2.2 Vector operations	7
2.2.1 Overloaded binary operations	7
2.2.2 Unary vector-valued operations	7
2.2.3 Norms and other reduction operations	8
2.3 Block vectors	8
2.4 Access to vector elements	8
2.5 Writing your own vector operations	9
3 Linear operators	9
3.1 Matrices	9
3.2 Implicit operators	9
3.2.1 Operator arithmetic	9
3.2.2 Transposition	10
3.2.3 Diagonal operators	10
3.2.4 Zero operators	10
3.2.5 Identity operators	10
3.2.6 Implicit inverses	10
3.3 Writing your own operator type	10
3.4 Block operators	10
3.5 Access to matrix data	10
3.5.1 Access to the nonzeros in a row	11

3.5.2	Access to values on the diagonal	11
3.5.3	Getting a lumped diagonal	11
3.6	Matrix-matrix operations	11
3.6.1	Matrix-matrix products	11
3.6.2	Matrix-matrix sums	11
3.6.3	Explicit diagonal matrix formation	12
4	Linear solvers	12
4.1	Preconditioners	12
5	Eigensolvers	12
6	Nonlinear solvers	12
6.1	Using a NOX nonlinear solver	12
6.2	Writing your own nonlinear solver	12
A	Review of design concepts	12

1 Overview

Most of the work in scientific computing involves operations with matrices and vectors. If you’ve programmed in Matlab, you’ll have learned that it’s a good idea – for both efficiency and human readability – to work with matrices and vectors as “objects” rather than doing operations by looping over indices. There are no built-in matrix and vector types in C++, but one can write classes to represent matrices and vectors.

It’s important to understand that in PDE simulation we will be forming systems of linear equations that are both very large and very sparse. The data structures for the matrices and vectors involved are fairly complicated, and the best choice of solution algorithm will depend strongly on the specific problem. There are a number of subpackages within Trilinos for doing sparse data structures and sparse solves. These in turn depend on lower-level libraries for dense linear algebra (LAPACK and BLAS) and for parallel communication (MPI). To provide a consistent and convenient user interface we “wrap” those capabilities in a suite of higher-level objects. This three-layer structure is shown in figure 1. Most of the time you’ll work with the highest-level objects; occasionally you might need to delve into the Trilinos mid-level objects if you want some customized behavior. Should you need to work with the mid-level objects, all of the Trilinos libraries have Doxygen documentation available.

1.1 Principal user-level objects

- The `VectorType` class is responsible for creating `VectorSpace` objects of a specified size and distribution over processors. For example, the `EpetraVectorType` subclass specifies that vectors will be stored as Epetra data structures. Note that “vector type” is not a mathematical object in the sense that vector spaces and vectors are; rather, it is a concept that lets us specify what *low-level software implementation* of vectors, spaces, and operators will be used.
- The `VectorSpace` class is responsible for creating vectors. This is done using the `createMember()` function. `VectorSpace` and `VectorType` are both examples of the *abstract factory* design pattern; an abstract factory is a software design trick that provides a common interface for creating objects, specific implementations of which might be constructed in very different ways.

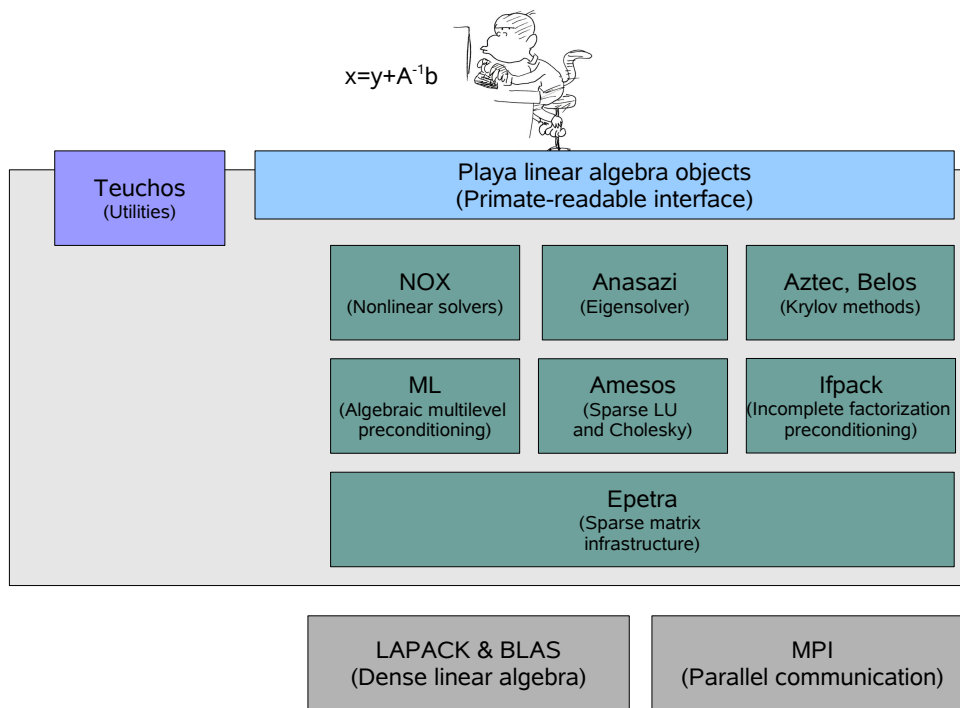


Figure 1: Schematic of relationship between high-level Playa linear algebra interface, mid-level Trilinos libraries for sparse linear algebra, and low-level BLAS, LAPACK, and MPI. The Playa linear algebra objects make it possible to write efficient code using high-level notation such as $x = y + A^{-1}b$.

- The `Vector` class represents vectors. Two compatible (*i.e.*, both from the same vector space) vectors can be added and subtracted with the `+` and `-` operators. The `*` operator between two vectors does the dot product. Other operations such as various norms, Hadamard products, various operations involving scalars, and element access are described in section 2.2.
- The `LinearOperator` class represents linear operators. However, many operators can be implemented “matrix free,” meaning it is not necessary to store any matrix elements. For example, while the identity operator has a matrix representation, it is inefficient to go through a matrix-vector multiplication when the assignment $x \leftarrow Iy$ can be effected by simply copying y into x . Thus, an identity operator isn’t implemented as a matrix, it’s simply an instruction to copy the input directly into the output.
- The `LinearSolver` class represents algorithms for solving linear equations.

All of these classes are templated on the scalar type, for instance, `Vector<double>` or `Vector<float>`. The Sundance PDE discretization capabilities are presently hardwired to double-precision real numbers, so we’ll use `Vector<double>` in all examples.

1.1.1 Handles, deep and shallow copies

The five principal classes, `VectorType`, `VectorSpace`, `Vector`, `LinearOperator`, and `LinearSolver` are all implemented as *reference-counted handles*. An important consequence of that fact is that copies are “shallow”. That means that an assignment such as

```
Vector<double> x = someSpace.createMember();
Vector<double> y = x;
```

does not create a new copy of the vector x , complete with new data. Rather, it creates a new “handle” to the same data. One advantage of this is obvious: vectors can be large, so we want to avoid making unnecessary copies. But note that any modification to y will also trigger the same modification to x , because x and y are referring to *exactly the same data in memory*. The potential for confusion and unintended side effects is obvious. Less obvious is that in certain important circumstances, such side effects are exactly what is needed for a clean user interface to efficient low-level code.

Should you want to make a “deep” copy of a vector, in which the copy has its own data and is fully independent of the original, use the `copy()` member function.

```
Vector<double> x = someSpace.createMember();
x.setToConstant(1.0);
Vector<double> y = x;
Vector<double> z = x.copy();
```

At this point x and y are two “handles” to the same underlying vector, whereas z is a vector that has, for now, the same elements as x . Now modify x

```
x.setToConstant(5.0);
```

and print norms of the three vectors

```
Out::root() << "||x|| = " << x.norm2() << endl;
Out::root() << "||y|| = " << y.norm2() << endl;
Out::root() << "||z|| = " << z.norm2() << endl;
```

The norms of x and y are consistent with the updated value even though the variable y has never been *directly* modified. The norm of z remains consistent with the original value of the vector x , because modifications to x are not propagated to its deep copies.

1.1.2 Example: A conjugate gradient solver

Here we show how these objects are used to write a simple conjugate gradient algorithm and apply it to finite-difference solution of the Poisson equation in 1D. The creation of the finite-difference matrix is assumed to be done by a function `buildFDPoisson1D()`. We don't show the details of the matrix creation function here; filling matrices is low-level code that Sundance will usually hide from you.

```
int numPerProc = 20;

VectorType<double> vecType = new EpetraVectorType();

LinearOperator<double> A = buildFDPoisson1D(vecType, numPerProc);

Out::root() << "Matrix A = " << endl; // print header on root processor only
Out::os() << A << endl;               // print matrix data on all processors
```

Having created the matrix, we now make a vector of compatible size and fill it with values. We'll use this as the RHS of $Ax = b$.

```
VectorSpace<double> space = A.domain();
Vector<double> b = space.createMember();

b.setToConstant(1.0);

Out::root() << "Vector b = " << endl; // print header on root processor only
Out::os() << b << endl;               // print matrix data on all processors
```

Now we write the CG algorithm. For simplicity, error checking is omitted.

```
Vector<double> x = b.copy(); // NOT x=b, which would be a shallow copy
Vector<double> r = b - A*x;
Vector<double> p = r.copy();

double tol = 1.0e-12;
int maxIter = 100;

Out::root() << "Running CG" << endl;
Out::root() << "tolerance = " << tol << endl;
Out::root() << "max iters = " << maxIter << endl;
Out::root() << "_____ " << endl;
for (int i=0; i<maxIter; i++)
{
    Vector<double> Ap = A*p; // save this, because we'll use it twice
    double rSqOld = r*r;
    double pAp = p*Ap;
    double alpha = rSqOld/pAp;

    x = x + alpha*p;
    r = r - alpha*Ap;

    double rSq = r*r;
    double rNorm = sqrt(rSq);
    Out::root() << "iter=" << setw(6) << i << setw(20) << rNorm << endl;

    if (rNorm < tol) break;

    double beta = rSq/rSqOld;
    p = r + beta*p;
}
```

Upon exiting the CG loop, print the solution.

```
Out::root() << "Solution: " << endl;
Out::os() << x << endl;
```

An industrial-strength CG solver would need preconditioning and error checking, and could be packaged up as a `LinearSolver` object.

1.1.3 Example: Inverse power iteration

Next we show code for calculation of the lowest eigenvalue of a matrix A by applying the power method to A^{-1} . The most important feature to look for in this example is the use of an implicit inverse operator. It's rarely a good idea to compute the matrix A^{-1} , so we want to avoid that. Instead, we create an *implicit inverse operator* that evaluates $A^{-1}y$ by using a `LinearSolver` object to solve the system $Ax = y$. Linear solvers are complicated enough that we'll usually build them by reading parameters from a file.

```
#include "Playa.hpp"
#include "FDMatrixPoisson1D.hpp"

int main(int argc, char** argv)
{
    try
    {
        Playa::init(&argc, &argv);

        int numPerProc = 1000;
        VectorType<double> vecType = new EpetraVectorType();
        LinearOperator<double> A = buildFDPoisson1D(vecType, numPerProc);
        VectorSpace<double> space = A.domain();

        Vector<double> x = space.createMember();
        x.setToConstant(1.0);

        LinearSolver<double> solver =
            LinearSolverBuilder::createSolver("amesos.xml");
        LinearOperator<double> AInv = inverse(A, solver);

        int maxIters = 100;
        double tol = 1.0e-12;
        double mu;
        double muPrev = 0.0;

        for (int i=0; i<maxIters; i++)
        {
            Vector<double> AInvX = AInv*x;
            mu = (x*AInvX)/(x*x);
            Out::os() << "Iter " << setw(5) << i
                << setw(25) << setprecision(10) << mu << endl;
            if (fabs(mu-muPrev) < tol) break;
            muPrev = mu;
            double AInvXNorm = AInvX.norm2();
            x = 1.0/AInvXNorm * AInvX;
        }
        Out::root() << "Lowest eigenvalue "
            << setw(25) << setprecision(10) << 1.0/mu << endl;
    }
    catch(exception& e)
```

```

{
    Playa::handleException(e);
}
Playa::finalize();
}

```

Both examples have concentrated on the *use* of matrix and vector objects rather than the *creation* of these objects. Sundance will automatically build matrices for rather complicated problems, and you'll rarely need to create matrices yourself. However, in writing advanced preconditioning, optimization, and solver algorithms you'll need to compose implicit operations.

2 Vectors

We now move on from the high-level overview to a discussion of the types and capabilities of vectors.

2.1 Creation of vectors

You will rarely call a vector constructor directly; the reason for this is that different vector libraries have different data requirements making it difficult to . Instead, vectors are built indirectly by calling the `createMember()` member function of `VectorSpace`. Each `VectorSpace` object contains the data needed to build vector objects, and the implementation of the `createMember()` function will use that data to invoke a constructor call.

2.2 Vector operations

2.2.1 Overloaded binary operations

The standard binary operations $\mathbf{a} \pm \mathbf{b}$, $\alpha \mathbf{a}$, and $\mathbf{a} \cdot \mathbf{b}$ are implemented via operator overloading.

Left operand	Operator	Right operand	Return type	Restrictions
\mathbf{a}	$+$	\mathbf{b}	Vector	Operands must be members of the same vector space
\mathbf{a}	$-$	\mathbf{b}	Vector	Operands must be members of the same vector space
α	$*$	\mathbf{a}	Vector	
\mathbf{a}	$*$	\mathbf{b}	Scalar	Operands must be members of the same vector space

These operations can be combined in any way that makes mathematical sense, for example:

```
Vector<double> v = 2.0*a - 4.0*b + (a*b)*c;
```

Note that the operation $(\mathbf{a} \cdot \mathbf{b}) * \mathbf{c}$ obeys the rules of precedence, so that the vector \mathbf{c} is multiplied by the scalar $\mathbf{a} \cdot \mathbf{b}$.

2.2.2 Unary vector-valued operations

The following functions operate elementwise on a vector, returning a new vector as a result. The original vector is unchanged.

Operation	Notes
<code>reciprocal()</code>	If any element is zero, a runtime exception will be thrown
<code>abs()</code>	
<code>copy()</code>	Makes a "deep" copy of a vector

```
Vector<double> w = v.reciprocal();
```

2.2.3 Norms and other reduction operations

Operation	Notes
<code>norm1()</code>	Computes $\ x\ _1$
<code>norm2()</code>	Computes $\ x\ _2$
<code>normInf()</code>	Computes $\ x\ _\infty$
<code>max()</code>	Largest element of x
<code>min()</code>	Smallest element of x

If $x = \begin{pmatrix} -1 & 0 & 0 & 1 & 2 \end{pmatrix}$ then

```
Out::os() << x.norm1() << endl;    // prints 4
Out::os() << x.norm2() << endl;    // prints sqrt(6)
Out::os() << x.normInf() << endl;  // prints 2
Out::os() << x.max() << endl;    // prints 2
Out::os() << x.min() << endl;    // prints -1
```

Also, there are methods to return the location of the minimum and maximum (used in algorithms for constrained optimization, to find the nearest constraint). Write these up later.

2.3 Block vectors

To be written

2.4 Access to vector elements

Before working directly with vector elements, stop to ask whether it's really necessary. Most numerical algorithms don't require access to vector elements. For instance, a conjugate gradient solver uses only vector addition, multiplication of a vector by a scalar, and the inner product between vectors. All of these can be carried out using member functions of `Vector` with no need for element access. There are two cases in which you may need element access:

- If you need to write a new vector operation not already supported. In such cases, for efficiency and generality it is best in the long run to implement the operation using the reduction/transformation operator (RTOp) interface. However, writing the operation via vector elements is a workable expedient to get your code up and running quickly.
- If you need to load elements into a vector, for instance when reading from a file.

If vector element access really is what you need, then here's how to do it. An element of a vector is identified uniquely by its *global index*. To get or set the value of that element, use the `getElement()` and `setElement()` functions as follows.

```
/* set vec[10] = 3.14; */
vec.setElement(10, 3.14);

/* view the new element */
Out::os() << vec.getElement(10);
```

In a parallel program with a distributed vector space, each processor will contain only a subset of the global indices. To find the range of global indices on each processor, you can use methods of `VectorSpace`. For example, the following code when run in SPMD mode will fill a distributed vector with values $x_n = \sqrt{n}$. Each processor sets only the elements "living" on that processor.

```
int low = vec.space().lowestLocallyOwnedIndex();
int high = low + vec.space().numLocalElements();
for (int g=low; g<high; g++) vec.setElement(g, sqrt(g));
```


Alternatively, you can use a `SequentialIterator` object to walk in order over vector elements. For example, this code fills $x_n = \sqrt{n}$ using iterators.

```
SequentialIterator<double> iter;
for (iter=vec.space().begin(); iter != vec.space().end(); iter++)
{
    int g = iter.globalIndex();
    vec[iter] = sqrt(g);
}
```

2.5 Writing your own vector operations

Need a clean interface to Bartlett's RTOp system.

3 Linear operators

Linear operators are represented by the `LinearOperator` class.

3.1 Matrices

Construction of matrices is a several-step process.

1. Create a `MatrixFactory` object
2. Call member functions of the `MatrixFactory` to configure the sparsity structure of the matrix
3. Call the `createMatrix()` member function of the `MatrixFactory` to allocate the matrix
4. Call member functions of the `LoadableMatrix` interface to set values of the nonzero elements.

If this seems complicated, it's because working with sparse matrices *is* complicated, and different sparse matrix implementations (there are several just in Epetra) need to be constructed in different ways. The `MatrixFactory` interface lets us hide the complexity of sparse matrix construction behind a common interface.

3.2 Implicit operators

We will frequently need to build operators out of simpler operators, without explicitly forming matrices. Several common types of implicit operators are described here.

In the following examples it is assumed that operators A and B have been created by some function.

```
LinearOperator<double> A = makeSomeMatrix();
LinearOperator<double> B = makeSomeOtherMatrix();
```

3.2.1 Operator arithmetic

The action of a composed operator ABx is computed implicitly by first computing $y = Bx$, then computing Ay . There is no need to form the matrix AB . Similarly, $(A \pm B)x$ can be evaluated implicitly by computing $y = Ax$, $z = Bx$, then doing $y \pm z$. Action of a scaled operator αAx is done implicitly as $\alpha(Ax)$. Any combination of these can be specified using overloaded operators, for example:

```
LinearOperator<double> C = A + B;
LinearOperator<double> D = 2.0*A - 0.5*B + 1.2*C;
LinearOperator<double> E = A*B;
```

3.2.2 Transposition

Most good sparse matrix packages have the ability to compute $A^T x$ without explicitly forming A^T . Given that, together with implicit composition, we can do $(AB)^T x = B^T A^T x$ implicitly as well, and with implicit addition we can do $(A \pm B)^T x = A^T x \pm B^T x$. The `transposedOperator()` function creates an operator object that knows to apply these rules.

```
LinearOperator<double> At = transposedOperator(A);
```

3.2.3 Diagonal operators

A diagonal operator can be represented with nothing but a vector of diagonal elements. To make a diagonal operator, call the `diagonalOperator()` function with the vector to be put on the diagonal.

```
Vector<double> d = makeSomeVector();  
LinearOperator<double> D = diagonalOperator(d);
```

3.2.4 Zero operators

Application of the zero operator returns the zero vector of the range space of the operator. To make a zero operator, call the `zeroOperator()` function with arguments that specify the domain and range spaces of the operator.

```
LinearOperator<double> zero = zeroOperator(domain, range);
```

3.2.5 Identity operators

The identity operator simply returns a copy of the operand.

```
LinearOperator<double> I = identityOperator(space);
```

3.2.6 Implicit inverses

The operation $y = A^{-1}x$ is computed implicitly by solving the system $Ay = x$. It is necessary to specify the solver algorithm that will be used to solve the system.

```
LinearOperator<double> AInv = inverse(A, solver);
```

3.3 Writing your own operator type

Write your class to conform to the `SimplifiedLinearOpBase` abstract interface, which basically means writing a member function that applies the operator to a vector. *To be written.*

3.4 Block operators

To be written

3.5 Access to matrix data

You usually don't want to do this! Two exceptions are: row access for creating certain preconditioners, and access to the diagonal for diagonal preconditioning and other tricks.

3.5.1 Access to the nonzeros in a row

3.5.2 Access to values on the diagonal

You can extract the diagonal from an operator A that is stored in Epetra form

```
Vector<double> d = getEpetraDiagonal(A);
```

3.5.3 Getting a lumped diagonal

A lumped diagonal for A is a diagonal matrix D where $D_{ii} = \sum_{j=1}^N A_{ij}$. This can be done with high-level operations, as follows:

```
Vector<double> ones = A.domain().createMember();  
ones.setToConstant(1.0);
```

```
Vector<double> rowSums = A*ones;
```

```
LinearOperator<double> lump = diagonalOperator(rowSums);
```

Because this involves multiplications as well as addition of matrix elements, it is very slightly less efficient than a specialized row sum function. However, not all sparse matrix implementations will have a row sum function, but all will have a matrix-vector multiply, so the multiplication method is a simple solution.

This method of finding lumped diagonals is a special case of a more general idea called probing, in which properties of a matrix are determined, or at least estimated, through multiplications with a strategically-chosen sequence of vectors.

3.6 Matrix-matrix operations

3.6.1 Matrix-matrix products

Multiplication of matrices is expensive, and explicit calculation of AB should almost always be avoided if an implicit calculation of ABx will suffice. However, in some cases there is no alternative but to form AB explicitly. The function `epetraMatrixMatrixProduct` will multiply two Epetra CRS matrices, returning the result (also stored as an Epetra CRS matrix) as a `LinearOperator`. This example does the operation $C = AB$.

```
LinearOperator<double> C = epetraMatrixMatrixProduct(A, B);
```

An important special case where some performance optimizations are possible is that of multiplication of a matrix A with a diagonal matrix D . The product DA is equivalent to scaling the rows of A by the corresponding diagonal entry of D , whereas the product AD scales the columns. The diagonal matrix D can be represented by a vector d containing its diagonal elements. The computation of DA is done as shown:

```
LinearOperator<double> DA = epetraLeftScale(d, A);
```

The matrix A is unchanged. Calculation of AD is, similarly,

```
LinearOperator<double> AD = epetraRightScale(A, d);
```

3.6.2 Matrix-matrix sums

Explicit matrix-matrix addition is done with the function `epetraMatrixMatrixSum` as follows:

```
LinearOperator<double> APlusB = epetraMatrixMatrixSum(A, B);
```

The two operands must have compatible shapes, but need not have the same sparsity graph.

3.6.3 Explicit diagonal matrix formation

If for some reason you need to work with an explicit Epetra matrix representation of a diagonal operator, you can create one from a Vector d as shown

```
LinearOperator<double> D = makeEpetraDiagonalMatrix(d);
```

4 Linear solvers

A linear solver is an object that takes a matrix A , a vector b , and solves the equation $Ax = b$. There are many methods for solving systems, and even given the same method we might have several different choices of implementations of that method. The Trilinos library, for instance, has three major linear solver packages

- Amesos is a collection of direct (sparse LU and sparse Cholesky) solvers suitable for PDE problems up to a few hundred thousand unknowns.
- Belos is a collection of Krylov methods
- Aztec is a legacy collection of Krylov methods, being superseded by Belos. Aztec is (at present) somewhat faster than Belos, but is more difficult to customize to problems involving block structure, unusual preconditioners, or unusual stopping conditions.

We might want to use any one of these and it needs to be convenient to switch from one solver to another. The most common method of creating a solver is to read parameters from an XML file and invoke the `createSolver()` static method of the `LinearSolverBuilder` class.

Unfortunately, at present the solver parameters are not well documented. In some solver packages they're not documented at all outside the source code. We need to work with the Trilinos solver developers to improve this situation.

4.1 Preconditioners

5 Eigensolvers

To be written

6 Nonlinear solvers

6.1 Using a NOX nonlinear solver

6.2 Writing your own nonlinear solver

A Review of design concepts

Some important points to take away from this description of linear algebra objects are:

1. We try to hide complicated, implementation-dependent operations (such as configuration and filling of a sparse matrix) behind abstract factory interfaces.
 - (a) This lets us change implementations simply by changing which factory we use.
2. We do as few *explicit* matrix operations as possible. For example, in computing ABx or $A^{-1}x$ with overloaded operators we never actually form the matrices AB or A^{-1} . Explicit matrix-matrix products are available for those rare cases when they are needed.

3. We avoid low-level operations on vector *elements* whenever possible, preferring high-level functions that operate on the vector as an object.

Keep these considerations in mind as we move on to discuss objects for meshes, geometric regions, symbolic expressions, quadrature, and other components of a PDE simulation.