S Vectors

William W. Symes *

ABSTRACT

The S Vector class turns data sets stored in several common seismic disk formats into vector objects as defined in the Hilbert Class Library (HCL). Thus data stored in these formats may appear as input and/or output of algorithms coded in HCL, and processing streams based on these formats may include HCL applications. The class structure permits easy extension to many other forms of scientific data.

INTRODUCTION

A common source of difficulty in creating a scientific program is mismatch between the data structures of its component algorithms. Some algorithms have natural expressions in terms of abstract linear algebra and calculus in Hilbert space, with no reference needed to coordinates. Others involve explicit reference to coordinates, with data organized in multidimensional arrays. Scientific data, on the other hand, rarely consists of arrays of data samples alone: it almost always includes numeric and/or nonnumeric parameters other than data samples, encoding sample rates, equipment characteristics, and other experimental conditions in some way. Processing of such data absolutely requires access to these parameters.

Object oriented program design helps the scientific programmer bridge this diversity of data structures across many levels of abstraction. In this approach to program organization, *objects* encapsulate data together with code for operations on it. In the most widely used approach, *classes* organize objects sharing specific operations or properties, and subclasses *inherit* these properties. The Hilbert Class Library (HCL) provides an object hierarchy suitable for expression of many common scientific programming tasks. HCL includes definitions of vector spaces, vectors, linear and nonlinear functions, and a variety of algorithms written in terms of these basic objects. HCL is coded in C++, which is one of several computer languages supporting object orientation to some extent.

*The Rice Inversion Project, Department of Computational and Applied Mathematics, Rice University, Houston TX 77251-1892 USA, email `symes@caam.rice.edu`

For documentation on HCL, including download instructions and an informative paper on design and performance of the library, point your web browser at

`http://www.trip.caam.rice.edu`

and follow the HCL link.

I have designed the S Vector class to address the data structure diversity problem in the context of reflection seismology. S Vectors form a subclass of the general vector class in HCL (`HCL_Vector`), so can be used in any algorithm formulated in terms of `HCL_Vector`. S Vectors give the user uniform access to data stored in several seismic formats. The design hinges on the *key based data access* concept: data is organized in *records*, as in any flat-file database. Each record pairs a *parameter table* and a *data vector*. The parameter table stores numeric and non-numeric attributes, or parameters, of the data, in `key=value` pairs. The data vector stores the data samples themselves, and perhaps other information. The distinction is necessary in order to treat collections of records as vectors: algebraic operations may alter the data samples, but not the parameters. On the other hand two such collections may have identical attributes, so define vectors in the same vector space, but have different data samples. So membership in a space should depend on the parameters but not on the data samples.

A motivating example of this kind of data record is the SEGY trace, probably the most common seismic exchange format: it consists of a 240 byte *trace header* encoding various information about the data acquisition geometry and conditions, followed by an array of data samples. Other common storage formats for seismic data, such as SU (Seismic Unix, derived from SEGY) and SEP (Stanford Exploration Project) formats, have similar structure. [In fact, the name of this vector class is the only letter in common between SEGY, SEP, and SU]. This package permits you to manipulate data stored in these three formats as HCL vectors. You can apply to such data any algorithm coded in HCL, and you can include HCL applications in shell scripts using other packages such as SU.

While I have concentrated on seismic data formats, the concept is much more general. I've coded the key based data access concept in a superclass of S Vector. This superclass permits easy extension to any type of data structure conforming to the key based data access model. S Vector is appropriate for the conventional seismic formats mentioned in the last paragraph. Newer seismic formats, such as SEP90 and SU3D, are obvious candidates for specializations of the superclass, but log and topographic data, navigation data, and aggregates of all of these data types also fit into the key based data access framework.

This paper will explain how to write programs using S Vectors. The code is (or will eventually be) internally instrumented so as to produce detailed documentation of various implementation details, of the same type as is available for HCL. The following two sections present operational overviews of the S Space and S Vector classes respectively.

Installation is very simple and is covered in the fourth section. The last section gives a very brief overview of the implementation, and discusses possible extensions.

USING S Spaces

S Vectors belong to S Spaces. S Spaces collect the parameter tables of the various data records in a *list*. This section explains how to create and manipulate S Spaces.

There are two purposes to the division between spaces and vectors. The first is mathematical: vector spaces play at least two essential roles in the expression of algorithms, in the formulation of error checks (is this vector a member of the domain (space) of this operator?) and in the provision of workspace (create a work vector in a space). The second is completely practical: the space contains all of the information necessary to perform many essential operations on the data, but is usually a much smaller object than the complete data. Therefore operations such as sorting and extraction of subvectors can be performed largely or entirely on the space without expensive data motion: for such operations, the space plays the same role as constructs variously termed "sort buffer", "geometry metafile", "gridfile",...

You construct an S Space using a the name `fname` of a data file in one of the permitted formats:

```
sSpace sp(fname);
```

The constructor reads through the entire file, collecting parameters into a list of tables, one for each record. Thus construction can be a lengthy process, for large files. However subsequent manipulations generally involve no i/o and are very quick.

The filename should end with a suffix indicating the file format: `.sep`, `.su`, and `.segy` are examples, with obvious meaning. This kludge will have to do, until Matt Schwab's suggestion for MIME types is realized.

Stew Levin suggested the optional `type` argument, which overrides the suffix-based file format decision. That's convenient for folks whose file naming convention doesn't follow the pattern outlined in the last paragraph. So if your data set is in SEGY format but its filename does not end in `.segy`, use the constructor form

```
sSpace sp(fname,"segy")
```

The present state of the code does not include intelligent format recognition, so if your data is *not* really SEGY you'll probably see some unpredictable sort of error occur as a result of this instruction.

You can query an S Space for the type of file on which it was based *via* the `Suffix` method:

```
char * filetype = sp.Suffix();
```

You compare S Spaces for equality using the usual C comparison notation:

```
if (sp1 == sp2) {
   (... do something ...)
}
```

and

```
if (sp1 != sp2) {
   (... do something else...)
}
```

The parameters contained in the space's list of tables are either critical or noncritical: critical parameters are generally those defined by the disk data format used in construction (eg. SEGY headers), and must be equal, key by key and table by table, for the comparison test to succeed. The noncritical parameters include some added by the constructor, and parameters you, the user, might add using methods to be described shortly. These play no role in S Space comparison.

The *virtual constructor* is an extremely important tool in OO design: it permits allocation of workspace without specifying the precise type of the object occupying it at compile time. This trick is the key to abstract coding of algorithms. The virtual constructor of HCL vectors is a method of the corresponding space class ("give me a vector in this space"). You create a scratch or work S Vector from the S Space `sp` like so:

```
sVector * vec = (sVector *)sp.Member();
```

Note that the `Member()` function returns a pointer, and that it is a pointer to an `HCL_Vector`, which must be *cast* to correct its type to `sVector`.

A brief digression on memory management: HCL methods return *references* to objects when no dynamic memory allocation is involved in their creation. HCL methods return *pointers* to objects when these objects contain dynamically allocated memory, which you must deallocate explicitly in C++. We mean this convention to function as a reminder to programmers: when you are done with `vec` in your program segment, use the `HCL_delete` operator to free its memory space:

```
HCL_delete(vec);
```

[The Java port of HCL will render these distinctions meaningless: in Java, all memory is allocated from the heap, and is garbage-collected. All methods will return references.]

An S Space is really a list of "record spaces", each pairing a parameter table with a data space. In the current implementation, there is only one data space: all data records must belong to the same HCL vector space, and in particular have the same length. [It would be easy to introduce a generalization to variable length records, but the resulting code is intrinsically less efficient. Since the data I intend to handle in the near future has fixed length records, I've postponed writing the obvious generalization.]

At any point in a program involving an S Space, you are somewhere in its list of (parameter table, data space) pairs. The class provides three basic methods for navigating the list, namely `Rewind`, which positions you at the beginning, `Next`, which moves you forward one position, and `Seek`, which positions you at a specified record number. `Next` returns an integer, which is zero when you are at the end of the list, nonzero otherwise. So the standard code for iterating through an S Space `sp` is

```
sp.Rewind();
do {
  (... something ...)
} while (sp.Next());
```

`Seek` takes an integer argument which specifies the record number, with the first record numbered zero. You can determine the number of records using the `NumberOfRecords` function:

```
int n = sp.NumberOfRecords();
if (0 <= j && j < n) {
  sp.Seek(j);
  ...
}
```

`Seek` returns no value.

An S Space object provides access to its components, the parameter table and the data sample space. The `Parameters` method returns a reference (not a pointer!) to a table object. You read and write data to the table *by keyword* using the `GetValue` and `PutValue` methods. For example the following loop reads the parameter with key `WinePrice` into an integer, doubles it, and writes it back, for every record in the space:

```
sp.Rewind();
int price;
do {
  sp.Parameters().GetValue("WinePrice",price);
  price=2*price;
  sp.Parameters().PutValue("WinePrice",price);
} while (sp.Next());
```

Actually both `GetValue` and `PutValue` return integers, zero for normal completion, nonzero for an error. So safer code is for example

```
sp.Rewind();
int price;
do {
  if (sp.Parameters().GetValue("WinePrice",price)) {
    cerr<<"Error: Parameters().GetValue"<<endl;
    cerr<<"failed to read WinePrice from table"<<endl;
    sp.Parameters().Write(cerr);
    exit(1);
  }
  price=2*price;
  if (sp.Parameters().PutValue("WinePrice",price)) {
    cerr<<"Error: Parameters().PutValue"<<endl;
    cerr<<"failed to write WinePrice to table"<<endl;
    sp.Parameters().Write(cerr);
    exit(1);
  }
} while (sp.Next());
```

In the present implementation you may *not* add arbitrary key-value pairs to the parameter tables of an S Space constructed from an `su` or `segy` data file after its construction. The available keywords for such spaces are listed in `$KBDAROOT/s/include/suhdr.h`. You *may* add arbitrary key-value pairs to the parameter tables of an S Space constructed from an `sep` data file.

Data records are also HCL Vectors, so the second part of the (parameter table, data space) pair is an HCL vector space. For the current implementation of S Vector, these belong to the very simplest HCL vector space class, `HCL_RnSpace_s`, the space of simple single-precision floating point arrays. The HCL documentation contains detailed information on the structure and implementation of this class and its associated vector class, `HCL_RnVector_s`. Its only additional operation beyond comparison and the virtual vector constructor is the `Dim` method which returns the dimension.

Beyond being of fixed length, the data in each record of an S Vector package is *regularly sampled*. Note that this restriction applies only *within* the record: unlike (classic) SEPlib, and like SEGY and SU, S Vector permits the parameters to vary arbitrarily *between* records. To permit access to the regular sampling geometry within data records, the parameter table gets several parameters added to it on construction. These parameters describe the data record as a *regularly sampled hypercube* of arbitrary dimension, following SEPlib conventions; they are:

- `rd` - the dimension of the record hypercube;
- `nx, x=1,...,rd`: the number of samples along each axis;
- `ox, x=1,...,rd`: the coordinate of the first sample on each axis;
- `dx, x=1,...,rd`: the sample rate on each axis.

For S Spaces based on SEP format files, `rd` really is arbitrary, and is identical to the SEP header word with the same key, as is the case for the other keys listed above. See the current documentation on the `SGF` vector class for detailed discussion of the meaning of other SEP header words.

For spaces based on SU and SEGY data, on the other hand, the data records are seismic traces and are intrinsically one-dimensional, so `rd = 1`. Also `n1`, `o1`, and `d1` are aliases for SEGY header words:

- `n1` for `ns`;
- `d1` for `dt` (converted to ms or m);
- `o1` for `delrt` (in ms or m).

The `DataSpace` method returns the space of an individual record, as a reference to an `HCL_VectorSpace_s` which must be cast before it can be used. For example the following code grabs a work vector suitable to hold the data array of a record:

```
HCL_RnSpace_s & dsp = (HCL_RnSpace_s &)(sp.DataSpace());
HCL_RnVector_s * dvec = (HCL_RnVector_s *)(dsp.Member());
```

You define a subspace based on a range of key values by means of the `SubSpace` method:

```
HCL_VectorSpace_s * SubSpace(char * key, int LowerLimit, int UpperLimit);
HCL_VectorSpace_s * SubSpace(char * key, float LowerLimit, float UpperLimit);
```

The subspace returned by the `SubSpace` method consists of those records for which the value associated with the specified key tests between the specified limits. The limits may be floats or integers but must both be of the same type. The test is inclusive.

This method returns a complete, standalone S Space, so dynamic memory allocation is involved, so it returns a pointer rather than a reference, which should be deleted when you are done with it. Also the pointer it returns is of an abstract type (`HCL_VectorSpace_s`) so must be cast before use, just as was the case with the `DataSpace` method.

`SubSpace` may be called recursively to specify arbitrary logical subcubes of the original space.

For example to extract the `sSpace` of records with offsets between 125 and 2500 m, execute

```
int minoff=125;
int maxoff=2500;
sSpace * subsp= (sSpace *)(sp.SubSpace("offset",minoff,maxoff);
```

If the `SubSpace` method finds no records satisfying the test, it returns a `NULL` pointer. So a "safe" version of the last code fragment would be something like

```
int minoff=125;
int maxoff=2500;
sSpace * subsp = (sSpace *)(sp.SubSpace("offset",minoff,maxoff);
if (subsp == NULL) {
  (... bail out ...)
}
else {
  (... do something ...)
}
```

To assist in locating usable key ranges, the class provides direct access to limits through the `Min` and `Max` functions, which take a key as argument:

```
int minkey = sp.Min(key);
int maxkey = sp.Max(key);
```

return the minimum and maximum values associated to `key` in the S Space `sp`. Float valued methods of the same names overload these.

Several other methods are essentially system utilities, so not generally useful in programming. I refer the reader to the (as yet nonexistant) documentation for details of these.

USING S VECTORS

There are two S Vector constructors, one which accesses existing files ("old data") and one which creates new files ("new data").

The old data constructor takes a filename and an optional integer and string flags:

```
sVector vec(fname);
```

or

```
int incore=1;
sVector vec(fname,incore);
```

or

```
int incore=0;
char * type = new char[5];
strcpy(type,"segy");
sVector vec(fname,incore,type);
delete [] tyoe;
```

The files used in S Vector construction are the same as those used in S Space construction, and are subject to the same constraints. As before, construction involves reading the entire file (for SU and SEGY, though not for SEP), so can be lengthy for large data files.

The `type` flag (`"segy"` in the example above) is string valued and overrides the file format decision ordinarily based on the suffix. This works the same way as does the similar flag for S Space construction, following Stew Levin's suggestion. Note that you must explicitly include the `incore` flag in the parameter list in order to use the `type` flag.

The `incore` flag is an intrusion of the reality of memory hierarchy into the abstraction of the S Vector class. If this flag is unset (i.e. zero, the default), then the data will be read from and written to the source file at its natural level of granularity, namely the record. For SEGY or SU, this means the trace; for SEP, the definition of record is variable.

Setting the `incore` flag causes the entire data set to be read into a buffer allocated in core memory, and from then on until destruction access to the data is to this buffer. On destruction, the data in the buffer is written back to the file. In the present design there is no other synchronization point.

This implementation detail is an annoyance but addresses an absolutely essential efficiency issue: operations on the data buffer are far faster than operations involving disk i/o, so it is *always* better to set the incore flag if (a) lots of arithmetic operations are to be performed on the data, and (b) adequate memory is available. This design leaves the decision to the user as to whether adequate memory is available. The usual use of the incore feature will probably center around subvectors, as will be discussed at the end of this section.

The new data constructor creates a vector in a prescribed space:

```
sVector newvec(sp, fname, incore);
```

The last two arguments - the filename string `fname` and the incore flag integer `incore` - are defaulted (i.e. optional). The default value of `fname` is `NULL`, in which case the constructor grabs a temporary file from the system which is removed when the object is deleted. The S Space `Member` method uses this version of the S Vector constructor to create temporary workspace, and you can use it the same way. If the incore flag is set (by default, it is not) and `fname = NULL`, then no i/o at all takes place: you get a completely incore construction and destruction.

Note that the S Space argument `sp` already contains file format information, so no optional `type` argument appears in the new data constructor's parameter list.

You can convert an out of core data set to incore, and vis versa: the methods `setIncore` and `unsetIncore` (no arguments or return values) let you do this (example below). You can find out whether an S Vector is incore or not by the inquiry function `isIncore` which returns an integer interpreted as Boolean.

All vector operations defined in `HCL_Vector` are available to S Vectors. For example, to add S Vector `vec2` to S Vector `vec1`, overwriting the contents of `vec1`, you use the `Add` method:

```
vec1.Add(vec2);
```

See the documentation on `HCL_Vector` for a list of the standard vector operations and their actions.

As with all HCL vectors, an S Vector knows what vector space it belongs to. The `Space` method returns a reference to this, as an `HCL_VectorSpace`, which has to be cast before it can be used. So here is how you access the parameter `cdp` in the current record of `vec`:

```
int cdp;
((sSpace &)(vec.Space()).Parameters().GetValue("cdp",cdp);
```

This is a bit verbose, so S Vector provides a "nick-method":

```
int cdp;
vec.Parameters().GetValue("cdp",cdp);
```

accomplishes the same thing. Note that you should probably trap a nonzero return as explained in the previous section.

The `DataVector` method returns a reference to the data vector component, as an `HCL_RnVector_s`. So you can get at the data samples for example like this:

```
for (int i=0;i<vec.DataVector().Dim();i++) {
  vec.DataVector().Data()[i]=3.1415927;
}
```

Of course rather than a low level loop like this which breaks data encapsulation, it is better to use a class method, when there is a class method (of `HCL_RnVector_s`) that does what you want:

```
float pi=3.1415927;
vec.DataVector().Fill(pi);
```

does the same thing, i.e. fills the current record with π.

`Rewind`, `Seek`, and `Next` function in the same way as they do in the S Space class, except that `Next` takes an optional argument which tells it whether to write the current data record before reading the next; by default this flag is set. It is useful to unset this flag if the data functions as read-only, since then you avoid half of the i/o. Here is a fragment of code which prints out the norm of each record:

```
vec.Rewind();
int ir=0;
int no_save_on_next = 0;
do {
```

```
  cerr<<"Norm of record "<<ir<<" = "<<vec.DataVector().Norm()<<endl;
  ir++;
} while (vec.Next(no_save_on_next));
```

Corresponding to `SubSpace`, there is a `SubVector` method which returns the projection of a vector onto the subspace defined by a range of key values, or `NULL` if there are no records with key values in the range. For example

```
int minoff=125;
int maxoff=2500;
sVector * subvec= (sVector *)(vec.SubVector("offset",minoff,maxoff);
if (subvec == NULL) {
  (... error message and/or recovery...)
}
else {
  (... do something ...)
}
```

returns all of the traces in a SEGY or SU data set with offsets between 125 and 2500, as an `sVector`. This is a completely new vector, with new storage allocation, so the method returns a pointer. The data file is shared between the vector and its subvector, however: the subvector reads and writes to the same file. So updating the subvector will update the records in the vector out of which it's made. In this respect the subvector is *not* an independent copy of part of the vector, but rather a means of accessing that part.

Here is an example: suppose that you want to zero out the first and last groups of 20 traces in a 64 trace CMP, stored on disk in SU format and objectified by the S Vector `vec`. The following code does the job:

```
sVector * subvec;
// first 20 traces
int trmin=1;
int trmax=20;
subvec= (sVector *)(vec.SubVector("tracl",trmin,trmax);
if (subvec != NULL) {
  subvec->Zero();
  HCL_delete(subvec);
}
// last 20 traces
trmin=45;
trmax=64;
subvec= (sVector *)(vec.SubVector("tracl",trmin,trmax);
```

```
if (subvec != NULL) {
  subvec->Zero();
  HCL_delete(subvec);
}
```

It is a fatal error to attempt to access a subvector (or subspace) based on an undefined keyword: S Vector responds to such an attempt by exiting with a diagnostic message on `stderr`. If the keyword is defined but no records satisfy the test, the `SubVector` and `SubSpace` methods return `NULL` pointers of the appropriate type. You can easily trap this condition with code like that just presented.

Subvectors are the real target of the incore construction. I expect that data sets will typically be inconvenient to store in core, but that programs will often have an internal level of granularity larger than the single record. When lots of flops are spent on subvectors *and workvectors returned by* `Member` based on their subspace, incore realizations will be very powerful speed enhancers. This possibility depends on the *incore inheritance* property of `Member`: if `subvec` is incore, then `subvec.Space().Member()` is also incore. Here is a relatively serious application: suppose that `L` is a `HCL_LinearOp` implementing a linear operator of some sort, whose constructor takes the the domain = range space as argument, which is the space of a single CMP. Suppose also that you want to solve approximately linear systems involving `L`, one for each CMP in the data set, using conjugate gradient iteration. Code like the following would do this:

```
// extract the space of the data
sSpace & datasp=(sSpace &)data.Space();

// construct solution vector (in same space)
sVector soln(datasp,"soln.su")

// declare pointers to cdps
sVector * datacdp;
sVector * solncdp;

// loop over cdps
for (i=cdpbeg;i<=cdpend;i++) {

  // extract subvectors for individual cdps
  datacdp=data.SubVector("cdp",i,i);
  solncdp=soln.SubVector("cdp",i,i);

  // drag them incore
  datacdp->setIncore();
  solncdp->setIncore();
```

```
  // extract single cdp space
  cdpsp=(sSpace &)(cdp->Space());

  // construct operator on cdp
  YourKindaOp L(cdpsp);

  // construct CG algorithm object
  HCL_CG cg(parfile);

  // solve
  cg.Solve(L,*datacdp,*solncdp);

  // deallocate incore subvectors
  HCL_delete(datacdp);
  HCL_delete(solncdp);
}
```

There are two points to make about this code. First, the HCL implementation of conjugate gradient iteration constructs all of its internal workspace using the `Member` function of the space class - since the vectors carrying the individual cdps are incore-ized, after that point in the code their spaces carry this information, so `Member` will return incore vectors, use of which involves no i/o at all. Second, the destruction of the incore subvectors at the end of the block causes their data to be written to the files storing their parent vectors, which is exactly what should happen. [That's true for the solution - perhaps `setIncore` should take an optional `NoWriteOnDelete` argument to avoid useless writes for data that you want to treat as read-only.]

INSTALLATION

The basic installation is completely simple. You must have installed HCL, as per instructions. Then set the environment variable `KBDAROOT` to the fully qualified path of the `kbda` directory, enter `kbda/s`, and type `make`. I set `KBDAROOT` in my `.cshrc`, and you should probably do this also.

Eventually the S Vector package will become part of the standard HCL release, but for the time being I will maintain S Vector as an independent package, freely available by anomymous ftp in the same way as HCL.

YET TO DO

Current status (as of late July 1998): SU and SEGY based constructors are implemented, along with the features described in the last two sections. SU is fully functional, and completely independend of the CWP software package. The SEGY implementation is actually a layer over the SU implementation, treating the first 3600 bytes as a data space property - so copies, subvectors, etc. get the same binary and ebcidic headers as the file on which their prototype is based. This design must be a temporary expedient, as it does not permit any nontrivial use of the binary and ebcidic headers and also limits the user to IEEE trace format - you must convert the SEGY file to IEEE first, in other words.

The SEGY constructor will eventually read both IBM and IEEE floats as well as the other trace formats permitted by the standard, and allow the user to access and modify the binary and ebcidic headers.

The SEP implementation is next on the "to-do" list, and will permit the S Vector class to supersede the current SGF class. I've chosen the syntax and form of the methods to force as little recoding as possible, and indeed S Vector is very close to a functional extension of SGF.

In general I have implemented only the plainly necessary features: constructors, basic linear algebra, sequential record access, projection and injection (subspaces and subvectors) based on key ranges, and a little bit of memory management. One feature which I expect to add soon is *key based sorting*, with and without actual copying of data.

At the moment `sVector` subclasses `HCL_Vector_s`, the basic single precision HCL vector class. However `sVector` is built on top of `HCL_RnVector_s` which subclasses `HCL_EuclideanVector_s`, the "matlab" space in which array operations like elementwise multiply are defined. So it will take relatively little recoding to subclass `sVector` from `HCL_EuclideanVector_s` instead, and I intend to do that.

I'm open to suggestion regarding other useful utilities which S Vector should implement. However bear in mind that I do not intend to produce a data processing package - that's already been done. Part of the purpose of the S Vector design is to permit you to use SU or SEPlib or even proprietary packages like ProMAX to do much basic processing, and to use HCL applications "seamlessly" in the same projects without tedious (or impossible!) format conversion.