B.2.3

UCRL- 98057
PREPRINT

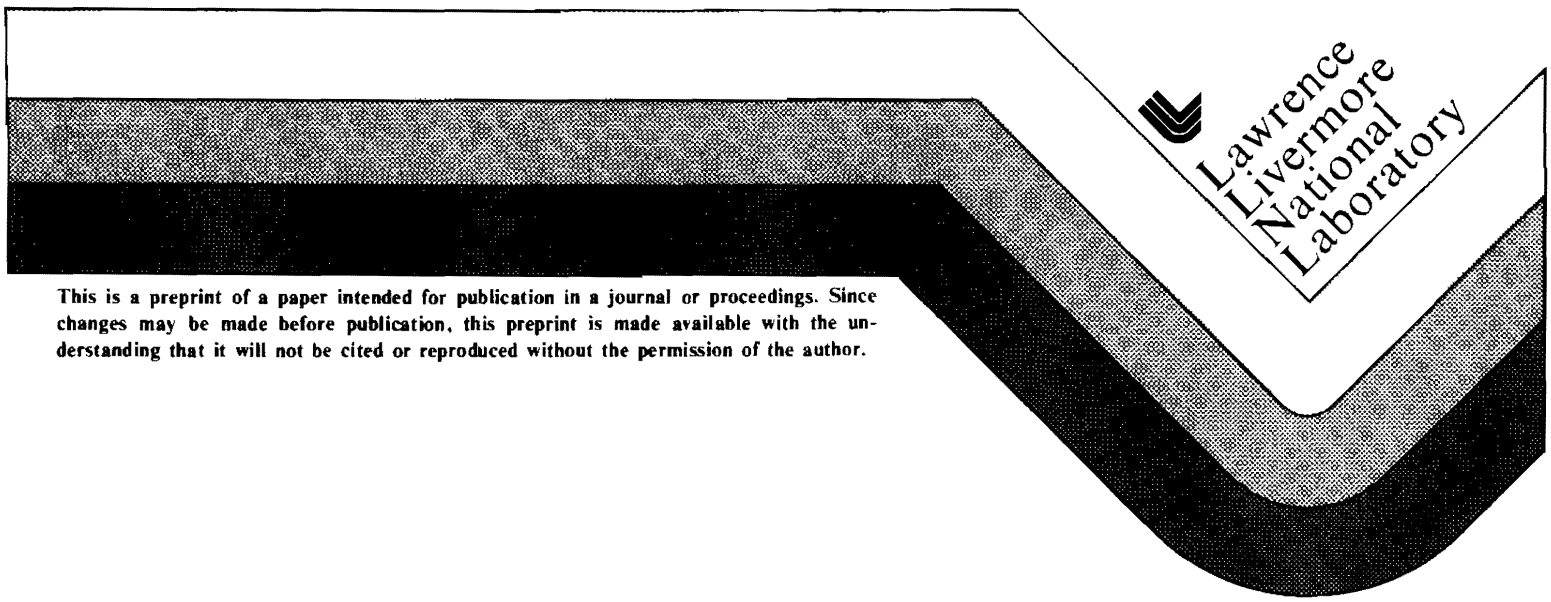
FIDIL: A LANGUAGE FOR SCIENTIFIC PROGRAMMING

Paul N. Hilfinger
University of California at Berkeley

Phillip Colella
Lawrence Livermore National Laboratory

This paper was prepared for submittal to
SIAM Frontiers in Applied Mathematics

January 1988



FIDIL: A LANGUAGE FOR SCIENTIFIC PROGRAMMING

PAUL N. HILFINGER* AND PHILLIP COLELLA†

Abstract. FIDIL is a new programming language for scientific computation. In this paper, we give a brief overview of the language, largely consisting of several extended examples from computational fluid dynamics.

1. Introduction. One fundamental goal of research in programming language design is to provide a better fit between problems and programming notation. In scientific computation, this quest is sometimes described as one of reducing the "semantic distance" between abstract mathematical descriptions of numerical methods and programs that implement them—in effect, of making abstract mathematical descriptions into programs. In its most ideal form, such a goal is generations beyond the current state of the art. However, there are intermediate points along the way to which we might aspire. In this paper, we describe one of them.

Currently, most numerical scientific programming is done in FORTRAN. This language has served its purpose well, but the basic operators, quantities, and definitional facilities that it supports are rather limited. Under the "FORTRAN model" of computation, programs consist of sequences of individual arithmetic operations on numbers contained in named scalar variables or in individual elements of arrays. FORTRAN provides a certain amount of abstraction in the form of subprograms, but these are sufficiently clumsy to use and define that in practice their application is limited (at least when measured against the practice in other programming languages). Despite these oft-cited limitations, the scientific community has largely adapted itself to FORTRAN, and has developed a large body of software in the form of libraries and application code.

Modern advances in numerical algorithms—motivated both by new applications and increased processing power—have led to increasingly complex programs, have made the task of converting algorithms to programs increasingly difficult, and thus have made it increasingly attractive to automate this conversion. One approach to automation is to provide a programming language that makes it easier to write about more complex, "bigger" entities: about operators on arrays rather than on individual array elements, about index sets with more general shapes than rectangles, and so forth. We have taken just this approach with the FIDIL (FInite DIfference) language, which we will describe in this article.

It is very difficult to evaluate a programming language, especially in the very early stages of its deployment. In order to give the reader some basis for judgment, we will present (in section 4) several realistic examples of FIDIL's use that are somewhat

* Department of Electrical Engineering and Computer Sciences, University of California at Berkeley. Research partially supported by the National Science Foundation under grant DCR-8451213.

† Lawrence Livermore National Laboratory. Research supported by U.S. Department of Energy, Office of Energy Research Applied Mathematical Sciences Program at the Lawrence Livermore National Laboratory under contract W-7405-Eng-48.

longer than is traditional for an overview paper about a programming language. We will start with a brief overview of FIDIL, first with a description of features common to most programming languages (section 2), and then with a description of features specifically intended for scientific computation (section 3).

The FIDIL system is still under development. Our experimental compiler is a Common Lisp program that runs on workstations and translates FIDIL into FORTRAN programs for a Cray X-MP. The use of FORTRAN as an intermediate language makes it relatively easy to use existing FORTRAN libraries, and may ease the task of porting the compiler to other machines.

2. Overview of general-purpose facilities. FIDIL, like any programming language built with a specific problem area in mind, comprises not only features specific to that area, but also a more general framework such as one might find in any language. We will begin by describing this framework.

2.1. General Program Structure. A FIDIL program consists of a sequence of declarations of constants and variables. Variable declarations use the "*type list-of-variable-names*" format of FORTRAN and the ALGOL family, as in

integer *x, y*;

Constant declarations all have the form "**let** $s_1 = d_1, s_2 = d_2, \ldots,$" which evaluates the d_i in order and defines the symbols s_i to have those values. As illustrated in the following example, constants can be ordinary scalar values, arrays (called *maps* in FIDIL; see section 3), functions, or types.

```
let
    n = 3,                      /* Scalar constant */
    V = [0, 0, 0, 1],           /* Array constant */
    rad = proc (real deg) -> real: 0.017453293 * deg,
                                /* A function */
    Point = struct [ real x, y ];
                                /* A type */
```

Declarations may also be nested inside function definitions, in which case, as is usual for languages in the ALGOL family, they apply only within that function and are computed once for each call of the function. To accommodate separate compilation, variables and certain constants can be *imported*—defined externally—or *exported* for import by other programs. In the case of variables, this capability corresponds to COMMON blocks in FORTRAN. The effect of an executable FIDIL program as a whole is defined as a call to a function with the distinguished name *main*.

Constant declarations and assignment-free expressions play an important role in FIDIL; the language has a distinct bias toward their use in places where programmers in FORTRAN, ALGOL, C, or Pascal might use sequences of assignments. It is not that such traditional programming is any more difficult in FIDIL than in FORTRAN, for example, but rather that it is less necessary. The bias takes the form of primitive operations for building arbitrarily complex objects in single expressions. As we shall illustrate in sections 3 and 4, the end result of this bias is that the execution of a

FIDIL program tends to consist of a sequence of assignment statements in which the computation of the value to be assigned is very large; that is, there is a great deal of computation between assignment statements. There are two reasons this effect is desirable. First, at an abstract level, the scientific programmer deals with "large" operations on large objects: decompositions of matrices, applications of difference operators to all values on a grid. It is only an artifact of conventional programming languages—inherited from underlying machine architectures—that such operations ultimately must be written as individual operations on scalar variables. Second, from the concrete level of the compiler, the analysis and transformation necessary to take advantage of pipelined or parallel machine architectures is generally easier when assignment statements are minimized.

FIDIL requires that all named entities be declared and that each declared entity have a single type, which may be indicated explicitly, as for variable declarations, or inherited from the entity's definition, as for constants. These types include the usual scalar types—integers, reals, complex numbers, booleans, and characters,—domain and map types (which encompass arrays), functional types, and record types. Since scalar types in FIDIL do not differ significantly from their realization in other languages, we shall say no more about them, and concentrate instead on domain and map types in section 3, functions and functional types in section 2.3, and record types in section 2.2.

2.2. Record Types. A record value (or variable), also called a record, is a collection of named values (or variables), called *fields*. Record types describe a class of records by giving the number, names, and types of the fields of all records in this class.

```
let
    State = struct [ real x, y, px, py, m ];

    State S;
```

The declarations above define *State* to be a record type whose values have four real fields, and then define S to be a *State* variable. S may be assigned to or passed as a parameter to subprograms, just as for objects of scalar types. The individual fields of S are accessible using *field selectors*, which are notated with a functional syntax as in the following example.

```
x(S) := px(S) / m * dt + x(S);
```

In keeping with previous comments about the infrequency of assignment, a FIDIL programmer might write the following instead.

```
S := State[ px(S)/m * dt, py(S)/m *dt, px(S) + fx*dt, py(S) + fy*dt, m(S)];
```

The right-hand side of this latter assignment statement illustrates the use of a *record constructor* to create an entire record at once.

2.3. Defining functions and operators. Earlier, we saw a definition of a simple function for converting degrees to radians.

```
let
    rad = proc (real deg) -> real: 0.017453293 * deg;
```

The syntax here is suggestive: it has the same form as the definition of a named constant, suggesting that the phrase to the right of the equals sign denotes a value in its own right. This is indeed the case; the expression defining the function *rad* is a *subprogram literal*. It has no name in isolation, but simply denotes "a function taking a single real parameter, call it *deg*, and returning the real value computed by the formula 0.017453293 * *deg*." The most common use for subprogram literals is in the context shown—as definitions of function names—but they are sometimes useful as anonymous function arguments to other subprograms and, as we shall see later, in defining functionals.

One prominent characteristic of mathematical notation is our tendency to reuse the same notation for multiple purposes. Programming languages present more opportunities for such reuse, since they tend to introduce mathematically artificial distinctions—as between "short real" numbers and "long real" numbers. FIDIL allows the *overloading* of notation so that a conventional or suggestive name may be used wherever it is appropriate. Hence, the definition of *rad* above may be extended to cover long real numbers as well.

```
extend
    rad = proc (long real deg) -> long real: 0.0174532935199433 * deg;
```

The compiler determines the particular definition of *rad* to use by the context of its use.

Another characteristic of mathematical notation, as constrasted with many programming languages, is that function calls are notated not just with alphanumeric names, but also with other operators having a more varied syntax. To accomodate this, FIDIL allows the definition and overloading of infix (binary), prefix, and postfix operators as functions or procedures. We might, for example, extend addition to work on *State* variables, as defined above.

```
extend
  + = proc (State p1,p2) -> State :
    begin
      let
        mc = m(p1)+m(p2);
      return State[ (x(p1)*m(p1)+x(p2)*m(p2))/mc,
                    (y(p1)*m(p1)+y(p2)*m(p2))/mc,
                    px(p1)+px(p2), py(p1)+py(p2), m(p1)+m(p2) ];
    end;
```

Besides showing the extension of '+' to *States*, this example illustrates a few minor points of syntax: the use of **begin** and **end** to provide a way of grouping several declarations and statements into a single statement or expression, and the use of the exit construct **return** to indicate the value of a function.

One common form of function definition defines one function as a specialization of another with certain parameter values fixed. For example, the following two declarations are identical. The second uses a *partial function closure* to abbreviate the definition.

```
let
  f = proc (State p) -> Force: attraction(p0, p);
let
  f = attraction(p0,?);
```

Here, we assume that the function *attraction* is previously defined to compute the contribution to the force (gravitational or whatever) on its second argument due to its first. The notation *attraction*($p0$,?) denotes a function of one argument that uses *attraction* to compute its result, using $p0$ as the first argument.

2.4. Functionals. FIDIL has been designed to accomodate "functional" programming, in which the principal operations employed are the applications of pure (side-effect-free or global assignment-free) functions to structured data. As we shall see, this particular programming method makes heavy use of functions on functions.

Of course, most conventional programming languages, including FORTRAN. provide the ability to pass functions as arguments to other subprograms. FIDIL goes further and allows functions to be *returned* as well, and in general to be calculated with the aid of appropriate operators. As an example, consider the extension of the (by now much-abused) operator '+' to functions; the sum of two unary functions is a new unary function that produces the sum of these functions' values. It can be defined as follows.

```
let
  UnaryFunction = proc (real x) -> real;
extend
  + =
    proc (UnaryFunction f1, f2) -> UnaryFunction:     /* (1) */
      proc (real y) -> real: f1(y) + f2(y);           /* (2) */
```

The fragment above first defines *UnaryFunction* as a mnemonic synonym for

```
proc (real x) -> real
```

which is, in isolation, a type describing values that are "procedures taking a single real arguments and returning a real result." Next, the subprogram literal giving the value of '+' indicates that '+' is a binary operator on unary functions $f1$ and $f2$—line (1)—and that its value is the subprogram that takes a real argument, x, and returns the sum of $f1$ and $f2$ at y—line (2).

2.5. Generic subprograms. As it stands, the definition of '+' in section 2.4 works only for functions on **real** values. A definition of precisely the same form makes perfect sense for functions of any numeric type, however. FIDIL provides a notation whereby a single *generic* subprogram declaration can serve essentially as the template

for an entire family of specific subprogram declarations. Thus, we can generalize the addition of functions as follows.

extend
$+$ = **proc** (**proc** (?T x) -> ?T $f1$, $f2$) -> **proc** (?T x) -> ?T:
proc (T y) -> T: $f1(y) + f2(y)$;

Here, the notation '?T' indicates a pattern variable for which any type may be substituted. This definition of '+' applies to any pair of (unary) functions on the same type, T, producing another function on T. The resulting function uses whatever definition of '+' is appropriate for values of type T. The actual rules here are somewhat tricky, since it is possible in principle to have the definition of '+' on T differ from place to place in a program. For the purposes of this paper, we shall simply assume that this situation does not occur and not go into the specific rules governing the selection of '+', on the general assumption that an unhealthy preoccupation with pathologies makes for poor language design.

2.6. Standard control constructs. FIDIL's constructs for conditional and iterative execution differ only in syntax from those of other languages. Figure 1 illustrates both in two fragments showing a sequential and then a binary search. In each case, the search routine accepts a one-dimensional array with a least index of 0 and a value to search for in the array, returning either the index of the value in the array, or -1 if the value does not appear.

The **if-elif-else-fi** construct, taken directly from Algol 68, allows the programmer to indicate a sequence of conditions and the desired computations to be performed under each of those conditions. It may be used either as a statement, to indicate which of several imperative actions to take, or as an expression, to indicate which of several possible expressions to compute. As we shall see in section 3.2, the conditional construct also extends to conditions that produce arrays of logical values, rather than single logical values.

The **do-od** construct indicates an infinite loop, which can be exited by an explicit **exit** or **return** statement (the latter causing exit from the enclosing subprogram as well.) A preceding **for** clause specifies an index set for the iterations. The fragment above illustrates a simple iteration by 1 through a range of integers. More general iterations are also possible. For example, one can iterate two variables over a rectangular set of integer pairs using the following construct.

for (i, j) **from** [1..N, 1..M] **do** ... **od**;

Here, the pairs are enumerated in row major order (j varies most rapidly). One can specify strides other than one, as in the following.

for i **from** [1..N] **by** 2 **do** ... **od**;

3. Domains and Maps. Two classes of data type, *domains* and *maps*, play a central role in FIDIL, because of their natural applications in algorithms that involve discretizing differential equations. Together, they constitute an extension of the array types universally found in other programming languages. An array in a language such as FORTRAN can be thought of as a mapping from some subset of $\mathbf{Z}^n$ (the set of

```
let
  search1 =
    proc (Vector A; integer x) -> integer:
      for i from [0 .. upb(A)] do
        if x = A[i] then return i;
        elif x > A[i] then return -1;
        fi;
      od,
  search2 =
    proc (Vector A; integer x) -> integer:
      begin
        integer i, j;
        i := 0;  j := upb(A);
        do
          if i >= j
            then exit;
          else
            let m = (i+j) div 2;
            if A[m] >= x then j := m;
            else i := m+1;
            fi;
          fi;
        od;
        return
          if j < i then -1
          elif A[i] = x then i
          else -1
          fi;
      end;
```

FIG. 1. *Two searches: search1 is linear and search2 is binary.*

n-tuples of integers) to some codomain of values. Unlike mappings that are defined as subroutines, arrays can be modified at individual points in the index set. The index set of a conventional array is finite, rectangular, and constant throughout the lifetime of the array. One typically denotes operations on arrays with dimension-by-dimension loops over the index set, indicating an action to be performed for each data value in the array as an explicit function of the index of that value.

Maps are the FIDIL data objects corresponding to arrays. Unlike conventional arrays, however, their index set need not be rectangular or fixed, and the primitive operations provided by FIDIL encourage the programmer to describe operations upon them with single expressions that deal with all their values at once, generally without explicit reference to indices. To accomplish this, the concept of array is split into that of a *domain*, which corresponds to an index set and contains tuples of integers, and of a *map*, which consists of a domain and a set of values, one for each element of the domain.

3.1. Domains and maps with fixed domain. We use the notation **domain**[n] to denote the type of an n-dimensional index set. A variable declared

domain[2] D;

can contain arbitrary sets of pairs of integers. A particular rectangular domain may be generated using a *domain constructor*, as in the following example, which sets D to the index set of an N by M FORTRAN array.

D := [1 .. N, 1 .. M];

Several standard set operations apply to domains; the operators '+', '-', and '*' denote union, set difference, and intersection, respectively. In addition, there are several operations, summarized in Table 1, that are appropriate for index sets.

For a domain D, the notation

[D] T X;

declares a variable X that is a map with index set D and element values of type T. The type T (the *codomain*) may be any type; it is not restricted to scalar types. As a shorthand, a simple rectangular domain constructor may also be used to denote the domain, as in the following.

[1 .. 10, 1 .. M] T Y;

In both these cases, the domain is evaluated at the time the variable is declared and remains fixed throughout the variable's lifetime.

The precise domains of formal parameters to procedures need not be specified, but may be supplied by the actual parameters at the time of call, as in FORTRAN. For example, the following header is appropriate for a function that solves a system of linear equations $Ax = b$.

let

solve = **proc** ([*2] **real** A; [] **real** b; **ref** [] **real** x) : ...

The notation '[*2]' indicates that the index sets of A is a two-dimensional domain whose contents is given by the arguments at the time of call. The notation '[]' is short for '[*1].' The dimensionality of the domain may be supplied by the call as

well. For example, the following header is appropriate for a function that finds the largest element in an array.

let

largest = **proc** ([*?*n*] **real** *A*) -> **real** : ...

Again, in all of these cases, the index set of the formal parameters so defined is taken to be fixed over the call.

3.2. Operations on maps. FIDIL provides for ordinary access to individual elements of a map. If d is an element of the domain of a map A, then $A[d]$ is the element of A at index position d. However, FIDIL encourages the programmer to avoid the use of this construct and to try to deal with entire arrays at once. There is an extensive set of pre-defined standard functions and constructs for forming map-valued expressions. This in turn makes it easier for the compiler to make use of implementation techniques, such as vectorization or parallel processing, that process the entire array efficiently.

As for arrays in most modern languages, entire FIDIL maps may be assigned in a single operation, as in

A := *map-valued expression*;

Likewise, constant map values may be defined in one declaration:

let *C* = *map-valued expression*;

Finally, there are various ways to assign to only a portion of a map. The statement

A *:= *E*;

for a map-valued expression E, assigns to only those elements $A[p]$ for which p is an element of both the domains of A and E.

The *map constructor* allows specification of the elements and index set of an entire map in one expression. The statement

V := [E_1, E_2, ...]

assigns to V a one-dimensional map such that $V[i] = E_i$. One may also use a rule to specify the elements of a map, as in the following.

V := [*i* **from** *domainOf*(*V*) : *f*(*i*)];

This assigns to V a map whose element with index i is $f(i)$. The function *domainOf* yields the domain of its argument (a map). In the case of a constant map, the "*i* **from**" phrase is superfluous and may be omitted, as in

[*domainOf*(*V*): 0.0];

which yields a map that is identically 0 on the domain of V.

A programmer can specify maps as *restrictions* of other maps or as *unions* of a set of maps having disjoint domains. The following definitions first use the restriction operator, **on**, to cause *inner* to be a copy of the portion of A whose indices are between 1 and 99 and *rim* to contain the values of B for the other indices of A. Finally, they define C to have the same interior as A and border as B, using the map union operator, '(+).'

let

A = [*p* **from** [0..100, 0..100] : *g*(*p*)],
B = [*p* **from** *domainOf*(*A*) : *h*(*p*)],

inner = *A* **on** [1..99, 1..99],
rim = *B* **on** (*domainOf*(*A*) − *domainOf*(*inside*)),
C = *A* (+) *B*;

One of the most important operations in FIDIL is the "apply all" functional, denoted by the postfix operator '@'. Suppose that *g* is defined as follows.

let
g = **proc** (**real** *x,y*) −> **real** : ...;

Then the expression '*g*@' denotes a function on maps whose codomain is **real** that yields a map whose codomain is **real**. Formally, for maps (or map-valued expressions) *A* and *B*, we have the following.

g@(*A,B*) ≡ [*i* **from** *domainOf*(*A*) * *domainOf*(*B*) : *g*(*A*[*i*], *B*[*i*])]

That is, *g*@(*A,B*) yields a map whose domain is the intersection of the domains of the maps *A* and *B* and whose value at each point in that intersection is found by applying the pointwise operator *g* to the values of *A* and *B* at that point.

For notational convenience, the standard arithmetic operators, comparison operators, and certain others are already overloaded to operate on maps, so that, for example, *A*+*B*, denotes a map whose elements are the sums of corresponding elements of *A* and *B*. These operators are also overloaded to take a scalar as one operand and a map as the other, with the usual meanings. Thus, 2.0 * *A* is the result of multiplying each element of *A* by 2.0.

The conditional expression also extends automatically to take **logical**-valued maps as its conditional test. For example, the following statement defines *A*[*i*] to be taken from *B* at all indices where *B*[*i*] is non-negative, and otherwise from *C*.

let *A* = **if** *B* >= 0 **then** *B* **else** *C* **fi**;

The expression *B* >= 0 evaluates to a map from the domain of *B* to logical values. For those indices at which *B* ≥ 0, *A* is defined to agree with *B*. For those indices at which *C* is defined and *B* < 0, *A* and *C* agree. Another way of expressing this is to use the standard function *toDomain*, which converts a **logical**-valued map to a domain containing exactly those indices at which the map has a true value:

let *A* = (*B* **on** *toDomain*(*B* >= 0)) (+) (*C* **on** *toDomain*(*B* < 0))

Many map operations of interest in finite-difference methods are functions of neighbors of a map element, and not just the element itself. To accommodate such operations, FIDIL extends the shift operations on domains (Table 1) to maps in the obvious way (see Tables 2 and 3). Thus, after the definition

let *C* = *A*<<[1,1];

the map *C* has the property that *C*[[2,2]] is equal to *A*[[1,1]], and in general, that

C[*p*] = *A*[*p* − [1,1]]

This last equivalence also illustrates the extension of arithmetic operators to maps: elements of a two-dimensional domain are pairs of integers, represented as one-dimensional maps of two integers such as *p*. The subtraction *p* − [1,1], is therefore the result of subtracting 1 from each element of *p*.

The apply-all and shift operators allow the succinct description of difference operators. For example, consider a map defined to sample the values of a function *f* at

Expression	Meaning
valtype(D)	For a domain D: shorthand for the type of the elements contained in D. If D is a **domain**[n] for $n > 1$, then **valtype**(D) is [1..n] **integer**. For $n = 1$, **valtype**(D) is **integer**.
$D_1 + D_2$	Union of D_1 and D_2.
$D_1 * D_2$	Intersection of D_1 and D_2.
$D_1 - D_2$	Set difference of D_1 and D_2.
p **in** D	A logical expression that is true iff p (of type **valtype**(D)) is a member of D.
lwb(D), *upb*(D)	For D a **domain**[n]: A value of type **valtype**(D) whose k^{th} component is the minimum (*lwb*) or maximum (*upb*) value of of the k^{th} components of the elements of D.
shift(D, S), D << S	Where S is of type **valtype**(D) (or **integer** if $n = 1$) and n is the arity of D: The domain $\{d + S \mid d$ **in** $D\}$.
shift(D)	Same as *shift*(D, $-$*lwb*(D)).
contract(D, S)	The domain $\{d$ **div** $S \mid d$ **in** $D\}$.
expand(D, S)	The domain $\{d * S \mid d$ **in** $D\}$.
accrete(D)	The set of points that are within a distance 1 in all coordinates from some point of D.
boundary(D)	*accrete*(D) $-$ D.

TABLE 1
Some standard operations on domains.

$N + 1$ discrete points between 0 and 1.

let *fhat* = [i **from** [0 .. N] : $f(i * (1.0/N))$];

Then a discrete approximation to the first derivative of f over this range is given by the following definition.

let *df* = (*fhat* $-$ (*fhat* << [1])) * N;

Working this out by hand will reveal that

df[1] = (*df*[1]$-$*df*[0])*N, *df*[2] = (*df*[2]$-$*df*[1])*N, ...

Because the domain of *fhat* does not contain $N + 1$ and that of *fhat* << [1] does not contain 0, the domain of *df* is [1..N].

3.3. Maps with flexible domains. Map variables need not have fixed domains. The following declaration indicates that the domain of *front*, a two-dimensional array of *States*, can vary over time.

flex [*2] *State front*;

Expression	Meaning
NullMap(*n, type*)	The null *n*-dimensional map with codomain *type*.
domainOf(*X*)	The domain of map *X*. May also be assigned to.
toDomain(*X*)	{ *p* **from** *domainOf*(*X*) : *X*[*p*] }, where *X* is a **logical** map.
image(*X*)	where *X* is a map whose codomain is of type [**n*] **integer**: the domain $\{d \mid X[p] = d \text{ for some } p\}$.
upb(*X*), *lwb*(*X*)	*upb*(*domainOf*(*X*)), *lwb*(*domainOf*(*X*))
X # *Y*	For *X* and *Y* maps such that *Y*'s codomain is **valtype**(*domainOf*(*X*)): a map object—assignable if *X* is assignable—such that $(X\#Y)[p] \equiv X[Y[p]]$. This is the composition of *X* and *Y*; its domain is $\{p \in \text{domainOf}(Y) \mid Y[p] \in \text{domainOf}(X)\}$.
shift(*X*,*S*), *X*<<*S* *shift*(*X*)	where *S* is a [1..*n*] **integer** (an integer for $n = 1$), with default value -lwb(*X*), and *n* is the arity of *X*: the map *X* # [*p* **from** *domainOf*(*X*): *p*−*S*]. The operator *shift*, as well as *contract* and *expand* below, yields an object that is assignable if *X* is assignable.
contract(*X*, *S*)	*X* # [*p* **from** *contract*(*domainOf*(*X*), *S*): *S***p*].
expand(*X*,*S*)	*X* # [*p* **from** *expand*(*domainOf*(*X*), *S*): *p*/*S*].
X **on** *D*	The map *X* restricted to domain *D*. Also assignable if *X* is.
X (+) *Y*	where $domainOf(X) \cap domainOf(Y) = \{\}$: the union of the graphs of *X* and *Y*, whose codomains must be identical and whose domains must be of identical arity.

TABLE 2
Some standard operations on maps, part 1.

Expression	Meaning
$concat(E_1,\ldots,E_n)$	Concatenation of the E_i. There must be a type T such that each E_i is either a 1-dimensional map with a contiguous domain and codomain T, or a value of type T (which is treated treated as a one-element map with lower bound 0). The result has the same lower bound as E_1 and a length equal to the sum of the lengths of the E_i.
$F@$	Assuming that F takes arguments of type T_i and returns a result of type T: the (generic) function extending F to arguments of type $[D_i]\ T_i$ and return type D, where the D_i are domains of the same arity and D is the intersection of the D_i. The result of applying this function is the result of applying F pointwise to the elements corresponding to the intersection of the argument domains.
$F<@>$	for F as above returning type T_1: The extension of F to arguments of types $[D_i]T$ as above, returning a value of type $[D_1]T_1$ defined by $F<@>(x_1,\ldots,x_n) = F@(x_1,\ldots,x_n)\ (+)\ (x_1\ \mathbf{on}\ (D_1 - D))$.
$trace(A,S)$	where S is an integer i_1 or $S = [i_1,\ldots,i_r]$, A is a map with a rectangular domain of arity n, $0 < r < n$, and $1 \le i_1 < \ldots < i_r \le n$: The map B of arity $n-r$ defined as follows. $B[j_1,\ldots,j_{i_1-1},j_{i_1+1},\ldots,j_n] = \sum_{k=lwb(A,i_1)}^{upb(A,i_1)} A[j_1,\ldots,j_{i_1-1},k,j_{i_1+1},\ldots,j_n]$. That is, indices i_j are replaced by an index variable k and summed, for each value of the other indices. To be well defined, the bounds of all the dimensions i_m of A must be identical. If $n - r = 0$, the result is a scalar.

TABLE 3
Some standard operators on maps, part 2.

```
let
    Unary = proc(?T x) -> ?T;

extend
    # = proc(Unary f1,f2) -> Unary:          /* f1 composed with f2 */
          proc(T x) -> T: f1(f2(x)),
    + = proc(Unary f1,f2) -> Unary:
          proc(T x) -> T: f1(x) + f2(x),
    * = proc(Unary f1,f2) -> Unary:
          proc(T x) -> T: f1(x) * f2(x),
    * = proc(Unary f; ?T a) -> Unary:       /* Scalar multiplication */
          proc(T x) -> T: f(x) * a,
    * = proc(?T a; Unary f) -> Unary:
          proc(T x) -> T: a * f(x),
    Id = proc(?T f) -> ?T: f;              /* Identity */

let
    /* Shift operators: for map A, (E1(k))(A) = A << [k,0] */
    E1 = proc(integer k) -> Unary: shift(? ,[k,0]),
    E2 = proc(integer k) -> Unary: shift(? ,[0,k]);

let
    Div = proc ([1..2] [*2] ?T x) -> [*2] ?T:
          (Id - E1(1))(x[1]) + (Id - E2(1))(x[2]);
```

FIG. 2. *A simple operator calculus.*

One can set the domain of such a map directly, as in the following assignment.

```
domainOf(front) := ...;
```

Alternatively, one can assign a map value with an arbitrary two-dimensional domain to *front* as a whole:

```
front := [ i from D : h(i) ];
```

An important use of flexible array types is in specifying "ragged" arrays. For example, consider the type *BinGrid* defined as follows.

```
let
    Bin = flex [] State;
    BinGrid = [D] Bin;
```

Each *Bin* has a one-dimensional domain that is independent of that of any other *Bin* in a *BinGrid*. As its name might suggest, such a data type might describe the state of a system of particles distributed by spatial position into a set of bins.

3.4. An illustrative operator calculus. Figure 2 displays a set of definition that we will use later. These extensions of arithmetic and other operators allow the succinct description of new difference operators.

4. Examples. In this section, we will describe the implementation in FIDIL of two sets of algorithms from numerical PDE's: particle methods for the vorticity formulation of Euler's equations, and finite difference methods for hyperbolic conservation laws. For each case, we will present a simple algorithm, and then use that simple algorithm as a building block for a more complex, but efficient algorithm for solving the problem. We will restrict our attention to problems in two space dimensions, since that is the case for which there are FORTRAN implementations for all of these algorithms.

4.1. Example 1: Vortex methods. Euler's equations for the dynamics of an inviscid incompressible fluid in two space dimensions can be written in terms of transport of a scalar vorticity ω as follows.

$$\frac{\partial \omega}{\partial t} + \mathbf{u} \cdot \nabla \omega = 0$$

$$\omega = \omega(\mathbf{x}, t) \in \mathbf{R}, \mathbf{x} \in \mathbf{R}^2$$

$$\mathbf{u} = K \star \omega = \int K(\mathbf{x} - \mathbf{x}')\omega(\mathbf{x}')d\mathbf{x}'$$

$$K = -\frac{\mathbf{x}^\perp}{2\pi|\mathbf{x}|^2}\ ,\ \mathbf{x}^\perp = (-x_2, x_1)$$

The evolution of the vorticity ω is given by advection by a velocity field $\mathbf{u}$ that is a non-local function of ω: $\mathbf{u} = K \star \omega = \nabla \times (\Delta^{-1}\omega)$. The velocity $\mathbf{u}$ satisfies the incompressibility condition $\nabla \cdot \mathbf{u} = 0$, so that the total vorticity in the system is conserved. If we consider Lagrangian trajectories $\mathbf{x}(t)$ satisfying

$$\frac{d\mathbf{x}}{dt} = \mathbf{u}(\mathbf{x}(t), t), \tag{1}$$

then the vorticity along those trajectories remains unchanged:

$$\frac{d\omega}{dt}(\mathbf{x}(t), t) = \frac{\partial \omega}{\partial t} + \frac{d\mathbf{x}}{dt} \cdot \nabla \omega = \frac{\partial \omega}{\partial t} + \mathbf{u} \cdot \nabla \omega = 0,$$

i.e., $\omega(\mathbf{x}(t), t) = \omega(\mathbf{x}(0), 0)$.

Vortex methods use a particle representation of the vorticity as its fundamental discretization (see Chorin [4]).

$$\omega(\mathbf{x}, t) \approx \sum_j \omega_j f_\delta(|\mathbf{x} - \mathbf{x}_j(t)|), \quad f_\delta(\mathbf{x}) = \frac{1}{\delta^2} f\left(\frac{|\mathbf{x}|}{\delta}\right).$$

The function f_δ is a smoothed approximation to a delta function, with

$$2\pi \int_0^1 f(r) r dr = 1, \quad f(r) \equiv 0 \text{ for } r > 1.$$

The discretized dynamics are intended to mimic the Lagrangian dynamics of the vorticity given by (1). In semidiscrete form, they are given by the following.

$$\begin{aligned} \frac{d\mathbf{x}_i}{dt} &= \mathbf{u}(\mathbf{x}_i) \\ \mathbf{u}(\mathbf{x}_i) &= \sum_j K_\delta(\mathbf{x}_i - \mathbf{x}_j)\omega_j \\ K_\delta(\mathbf{x}) &= K \star f_\delta(\mathbf{x}). \end{aligned} \tag{2}$$

Since f_δ is a function of $|\mathbf{x}|$ only, explicit formulas can be given for K_δ, for example, when f is a polynomial. In any case, $K_\delta = K$ if $|\mathbf{x}| > \delta$.

Given the formula (2) for the velocity field evaluated at all the particle locations, straightforward application of some explicit ODE integration technique, which would call a procedure to evaluation $\mathbf{u}$, will yield a discretization in time. In figure 3, we give a FIDIL program for evaluating the right-hand side of the ODE (2). The program is divided into two procedures. The first procedure *vortex_blob* evaluates the velocity field at an arbitrary point, $\mathbf{x} \in \mathbf{R}^2$: $\mathbf{u}(\mathbf{x}) = \sum_j K_\delta(\mathbf{x} - \mathbf{x}_j)\omega_j$. The cutoff function f_δ used here is due to Hald [5], and leads to a K_δ given by

$$\begin{aligned} K_\delta(\mathbf{x}) &= \frac{\mathbf{x}^\perp}{2\pi\delta^2} F(\frac{|\mathbf{x}|}{\delta}), \text{ if } |\mathbf{x}| < \delta \\ &= -\frac{\mathbf{x}^\perp}{2\pi|\mathbf{x}|^2} \text{ otherwise} \end{aligned}$$

where

$$F(r) = -36r^5 + 140r^4 - 196r^3 + 105r^2 - 14.$$

Vortex_blob takes as arguments x, the location where the velocity is to be evaluated, and *omega*, a one dimensional map containing the information describing the vortices. The map takes values of type *vortex_record*, a user-defined record type containing *position*, *velocity* and *strength* fields required to describe a single vortex. In writing *vortex_blob*, we have taken advantage of the fact that $F(1) = -1$ to compress the two cases in the definition of K_δ into a single expression. We use the operator *trace* to perform the sum in (2). Finally, the procedure *vorvel* uses *vortex_blob* to evaluate the velocity field induced at the locations of all the vortices. We use partial closure and function application to make the body of this procedure a single expression.

The principal difficulty with the algorithm described above is that the computational effort required to evaluate the velocities is $O(N^2)$, where N is the number of particles. Anderson [1] introduced a particle-particle, particle-mesh approximation to

```
external integer
   numvors;
external real
   twopi, delta;
let
   vortex_record =
      struct[[1 .. 2] real position, velocity; real strength],
   twoVector = [1 .. 2] real;

external [1 .. numvors] vortex_record vortices;

let vortex_blob = proc(twoVector x; [] vortex_record omega) -> twoVector:
begin

   let
      delx = x - position@(omega),

      distance_fcn = proc(twoVector x)->real:
         sqrt(x[1]**2 + x[2]**2),

      distance = distance_fcn@(delx),
      perp = proc(twoVector x) -> twoVector: [-x[2] ,x[1]],
      maxrdel = max(?,delta)@(distance),

      F = proc(real rd) -> real:
         (-14 + rd**2*(105 - rd*(196 - rd*(140 - 36*rd))));

      return trace(
            F@(maxrdel/delta) * perp@(delx) * strength@(omega)
               /(twopi*maxrdel**2),
            1);
end;

let vorvel = proc:
   velocity@(vortices) := vortex_blob(?,vortices)@(position@(vortices));
```

FIG. 3. *FIDIL program for velocity field evaluation.*

the $O(N^2)$ calculation, called the Method of Local Corrections (MLC). This algorithm is essentially linear in the number of particles, and does not introduce any additional loss of accuracy in the approximation to the underlying vorticity transport.

In the MLC algorithm, one introduces a finite difference mesh that covers the region containing the particles. Without loss of generality, we assume a square finite difference mesh covering the unit square, with mesh spacing $h = 1/M$ for some integer M, and satisfying $\delta < h$. We also assume that the particle locations $\mathbf{x}_j$ are all contained in a slightly smaller square $[Ch, 1-Ch] \times [Ch, 1-Ch]$, where $C \geq 1$ is in principle problem-dependent. In practice, satisfactory results have been obtained with $C = 2$. We also introduce the function $B : \mathbf{R}^2 \to \mathbf{Z}^2$ defined by $B(\mathbf{x}) = \mathbf{k}$ if

$$x \in [(k_1 - 1)h, k_1 h] \times [(k_2 - 1)h, k_2 h].$$

The algorithm is given as follows.

1). Compute $R : [1, \ldots, M]^2 \to \mathbf{R}^2$, a discrete approximation to $\Delta \mathbf{u}$ on the finite difference grid. If we denote by Δ^{fd} the 9-point discretization of Δ, then

$$\begin{aligned} (R^i)_{\mathbf{k}} &= \omega_i (\Delta^{fd} K^i)_{\mathbf{k}}, \quad \text{if } |B(\mathbf{x}_i) - \mathbf{k}| \leq C \\ &= 0, \quad \text{otherwise} \\ R &= \sum_i R^i \end{aligned}$$

Here, $(K^i)_{\mathbf{k}} = K_\delta(\mathbf{k}h - \mathbf{x}_i)$, the array of values of $K_\delta(\cdot - \mathbf{x}_i)$ projected over the finite difference grid, and $|\mathbf{l} - \mathbf{m}| = \max(|l_1 - m_1|, |l_2 - m_2|)$, $\mathbf{l}, \mathbf{m} \in \mathbf{Z}^2$. We are able to truncate R^i to be zero outside a finite radius because $\Delta K_\delta(\cdot - \mathbf{x}_i) = 0$ outside the disc $|\mathbf{x} - \mathbf{x}_i| < \delta < h$; thus, sufficiently far from $\mathbf{x}_i$, the truncation error in Δ^{fd} applied to K_δ is small, and is well-approximated by zero.

2). Solve $\Delta^{fd} \mathbf{u}^{fd} = R$ using a fast Poisson solver. The use of fast Poisson solvers on rectangular grids is standard, and we won't discuss it here. The only subtlety in the present case is that the boundary conditions required are "infinite domain" boundary conditions, i.e., a set of discrete boundary conditions corresponding to the infinite domain Green's function $G \star \rho = u$, $G(\mathbf{x}) = -\frac{1}{2\pi} \log(|\mathbf{x}|)$. However, this can be done by performing two fast Poisson solves with Dirichlet boundary conditions.

3). Calculate the velocity field at the particle locations. For the collection of particles contained in a given cell, this is done in two steps. First, the velocity induced by particles in nearby cells is computed using the direct N-body formula (2). Then the effect of all more distant particle is interpolated from the finite difference grid, having first corrected the values from $\mathbf{u}^{fd}$ to reflect the fact that the influence of the nearby particles has already been accounted for in the local N-body calculation.

$$\begin{aligned} \mathbf{u}(\mathbf{x}_i) &= \sum_{j : |B(\mathbf{x}_i) - B(\mathbf{x}_j)| \leq C} \omega_j K_\delta(\mathbf{x}_i - \mathbf{x}_j) \\ &\quad + I(\mathbf{x}_i; \tilde{\mathbf{u}}_{\mathbf{l}_1} \ldots \tilde{\mathbf{u}}_{\mathbf{l}_{l'}}) \end{aligned} \tag{3}$$

$$\begin{aligned} \mathbf{l}^1, \ldots, \mathbf{l}^s &= B(\mathbf{x}_i),\ B(\mathbf{x}_i) \pm [0,1],\ B(\mathbf{x}_i) \pm [1,0] \\ \tilde{\mathbf{u}}_{\mathbf{l}^s} &= \mathbf{u}^{fd}_{\mathbf{l}^s} - \sum_{j:|B(\mathbf{x}_i)-B(\mathbf{x}_j)|} \omega_i K_\delta(\mathbf{l}^s h - \mathbf{x}_j) \end{aligned}$$

Here, $I(\mathbf{x}; \tilde{\mathbf{u}}_{l^1} \ldots \tilde{\mathbf{u}}_{l^s})$ is calculated using complex polynomial interpolation in the plane.

$$\begin{aligned} I(\mathbf{x}) &= (\mathrm{Re}(\mathcal{I}(z)), -\mathrm{Im}(\mathcal{I}(z))) \\ \mathcal{I}(z) &= \sum_{q=0}^{4} a_q z^q, \quad z = (\mathbf{x}_1 + \sqrt{-1}\mathbf{x}_2) \end{aligned}$$

with the coefficients chosen such that

$$\begin{aligned} \mathcal{I}(z_s) &= \tilde{u}_{\mathbf{l}^s} - \sqrt{-1}\tilde{v}_{\mathbf{l}^s} \\ z_s &= (l_1^s + \sqrt{-1}l_2^s)h \end{aligned}$$

In figure 4 we give a FIDIL implementation of Anderson's algorithm for evaluating the velocity field induced by a collection of vortices on themselves. The procedure *MLC* takes as input the arguments *psi*, which contains the vortex data structure, and h, the finite difference mesh spacing. *Psi* is a two-dimensional map whose values are one dimensional maps of varying sizes. The values of the one-dimensional maps are of type *vortex_record*. Thus *psi* represents the collection of vortices sorted into finite difference cells: the domain of *psi* is the finite difference grid, and a vortex at position $\mathbf{x}_i$ is represented by an entry in *psi*[k] only if $\mathbf{x}_i = \mathbf{k}$. *MLC* evaluates the velocity fields induced by the vortices in *psi* on themselves and stores them in *velocity* field of each *vortex_record* in *psi*.

The first step of *MLC* is performed in (A)–(B). For each cell, the velocity field induced by the vortices in that cell is calculated for all the points in a domain large enough so that the nine-point Laplacian applied to the resulting map is defined on *D_C*. To define the finite difference Laplacian, we use the operator and shift calculus defined in figure 2 which has been included in the header file *operator_calculus.h*. Then Δ^{fd} is applied, and the result is used to increment *RHS*. The second step the call to *PoissonSolve* (C), which we take to be an externally defined procedure, to obtain $\mathbf{u}^{fd}$ defined on the grid *domainOf*(*psi*). The third step is performed in (D)–(E). For each cell $\mathbf{k}$, all of the vortices which will contribute to the sums in (3) for the vortices in *psi*[k] are gathered into a single map *psi_corrections*. Then the velocities of *psi*[k] are initialized with sum of the local N–body velocities. Finally, *u_fd_local*, the map containing the values of $\tilde{\mathbf{u}}$ are computed, and the interpolated velocities added to *velocity@*(*psi*[k]).

4.2. Example 2: Finite Difference Methods. A large class of time-dependent problems in mathematical physics can be described as solutions to equations of the form

$$\frac{\partial U}{\partial t} + \frac{\partial F}{\partial x} + \frac{\partial G}{\partial y} = 0$$

```
#include "operator_calculus.h"

let
   Fd_Values = [*2] twoVector;

let
   toComplex= proc(real a,b) -> complex: a + b*1i,
   iota = proc(domain[?n] D) -> [D] valtype(D): [i from D: i];

let PoissonSolve = proc(Fd_Values u) -> Fd_Values:
   external PoissonSolve;

let
    C = 2,
    D_C = [-C .. C, -C .. C],
    rhs_stencil = accrete(D_C),
    interpolation_stencil = Image([[0,0],[0,1],[1,0],[-1,0],[0,-1]]);

let MLC = proc(ref [*2] flex [] vortex_record psi):
begin
   let
      D = [i from domainOf(psi): length(psi[i]) /= 0];
   [domainOf(psi)] twoVector    RHS;

   RHS := [ domainOf(psi): [0.0, 0.0] ];                    /* (A) */

   let Laplacian = proc (Fd_Values u) -> Fd_Values:
         (4.*(E1(1) + E1(-1) + E2(1) + E2(-1)) - 20.0*Id +
          E1(1)#E2(1) + E1(1)#E2(-1) + E1(-1)#E2(1) + E1(-1)#E2(-1))(u)

   for k from D do
      let
         u_rhs_local = vortex_blob(?, psi[k])@(iota(rhs_stencil << k)*h);
      RHS := RHS + Laplacian(u_rhs_local)/(6.0*h**2)
   od;                                                      /* (B) */

   let u_fd = PoissonSolve(RHS,h);                          /* (C) */
```

FIG. 4. *FIDIL program for the Method of Local Corrections (Part 1 of 2).*

```
   for k from D do                                              /* (D) */
      flex [*] vortex_record psi_corrections;
      psi_corrections := NullMap(1,vortex_record);

      for j from D_C do
         psi_corrections := concat(psi_corrections, psi[k+j])
      od;

      velocity@(psi[k]) :=
         velocity@(psi[k]) + vortex_blob(?, psi_corrections)@(x@(psi[k]));
      u_fd_local :=
         u_fd << -k on interpolation_stencil
         - vortex_blob(?, psi_corrections)@
            (iota(interpolation_stencil << k)*h);
      velocity@(psi[k]) :=
         velocity@(psi[k])
         + interp(?, u_fd_local)@((position@(psi[k]) - k*h)/h);
   od;                                                          /* (E) */
end /* of MLC */;

let interp = proc(twoVector x; Fd_Values u_fd) -> twoVector:
begin
   let
      a = toComplex@(u_fd[][1],-u_fd[][2]),
      z = toComplex(x[1],x[2]),
      coef = [
            a[[0,0]],
            -(a[[1,0]] - a[[-1,0]] - 1i*(a[[0,-1]] - a[[0,1]]))*.25,
            (a[[-1,0]] + a[[1,0]] - a[[0,1]] - a[[0,-1]])*.25,
            -(a[[-1,0]] - a[[1,0]] + 1i*(a[[0,-1]] - a[[0,1]]))*.25,
            -(a[[0,0]] - a[[1,0]] - a[[-1,0]] - a[[0,1]] - a[[0,-1]])*.25
           ],
      pofz = coef[1] + z*(coef[2] + z*(coef[3] + z*(coef[4] + z*coef[5]))));

      return([realPart(pofz), imagPart(pofz)]);
end /* of Interp */;
```

FIG. 4. *FIDIL program for the Method of Local Corrections (Part 2 of 2).*

$$U = U(x,y,t) \in \mathbf{R}^M, F, G = F(U), G(U)$$

Such systems are known as conservation laws, since the equations are in the form of a divergence in space-time of (U,F,G). Hyperbolic refers to the fact that the underlying dynamics is given locally by the propagation of signals with finite propagation velocity; in particular, the initial value problem is well-posed. In general, F and G are nonlinear functions of U; for example, in the case of Euler's equations for the dynamics of an inviscid compressible fluid, if we denote the components of U by $U = (\rho, m, n, E)$, then the fluxes are given by

$$F(U) = (m, \frac{m^2}{\rho} + p, \frac{mn}{\rho}, \frac{m}{\rho}(E+p)),$$

$$G(U) = (n, \frac{mn}{\rho}, \frac{n^2}{\rho} + p, \frac{n}{\rho}(E+p)),$$

where the thermodynamic pressure p is given by

$$p = (E - \frac{m^2+n^2}{2\rho^2})(\gamma - 1).$$

A widely used technique for discretizing conservation laws is to use finite difference methods whose form mimics at a discrete level the conservation form of the differential equations.

$$(4) \qquad U_{i,j}^{n+1} = U_{i,j}^n + \frac{\Delta t}{\Delta x \Delta y}[\Delta y(F_{i+1/2,j} - F_{i-1/2,j}) + \Delta x(G_{i,j-1/2} - G_{i,j+1/2})]$$

Here Δt is a temporal increment, Δx and Δy are spatial increments, and n , i , j, are the corresponding discrete temporal and spatial indices. The discrete evolution (4) has a geometric interpretation on the finite difference grid. We interpret $U_{i,j}^n$ as the average of U over the finite difference cell $\Delta_{i,j}$,

$$\Delta_{i,j} = [(i-1/2)\Delta x, (i+1/2)\Delta x] \times [(j-1/2)\Delta y, (j+1/2)\Delta y]$$

$$U_{i,j}^n \approx \frac{1}{\Delta x \Delta y} \int_{\Delta_{i,j}} U(x,y,n\Delta t) dx dy,$$

and the evolution of U can be thought of as given by a flux balance around the edges of $\Delta_{i,j}$.

The first algorithm we consider is a variation on one of the first algorithms for conservation laws, the Lax-Wendroff algorithm. We use a two-step formulation of a type first introduced by Richtmyer [6]. Here, and in what follows, we take the spatial

grid to be square, i.e. $\Delta x = \Delta y = h$. The algorithm we will consider is given as follows.

$$
\begin{aligned}
(5) \qquad U_{i+1/2,j+1/2} &= \frac{1}{4}(U^n_{i,j} + U^n_{i,j+1} + U^n_{i+1,j} + U^n_{i+1,j+1}) \\
&+ \frac{\Delta t}{4h}(F(U^n_{i,j}) + F(U^n_{i,j+1}) - F(U_{i+1,j}) - F(U^n_{i+1,j+1})) \\
&+ \frac{\Delta t}{4h}(G(U^n_{i,j}) - G(U^n_{i,j+1}) + G(U_{i+1,j}) - G(U^n_{i+1,j+1}))
\end{aligned}
$$

$$F_{i+1/2,j} = \frac{1}{2}(F(U_{i-1/2,j+1/2}) + F(U_{i+1/2,j+1/2}))$$

$$G_{i,j+1/2} = \frac{1}{2}(G(U_{i-1/2,j+1/2}) + G(U_{i+1/2,j+1/2}))$$

It is clear from the above description that, if we want to evolve the solution on a finite rectangular grid for one time step, it suffices to provide additional solution values on a border of cells one cell wide all around the grid. A fairly general way to implement such boundary conditions is to provide a procedure $\phi(U, h)$ which returns U_B, the values required on the border of cells surrounding the grid where U is defined.

In figure 5 we give a FIDIL implementation of the Lax-Wendroff algorithm (4), (5). Again, we use the operator and shift calculus from figure 2 to implement the algorithm. We have split the implementation into two pieces. *LW_Flux* takes as input a map containing the values of U on the extended grid required to computed the fluxes $F_{i+1/2,j}$,$G_{i,j+1/2}$. It returns a map of type [1 .. 2] **flex** *Values* containing those fluxes. This map-valued map is a natural type for describing fluxes for conservation laws, since one needs a map of type *Values* with a different domain for the flux in each coordinate direction. The procedure *LW* calls *phi* to calculate the boundary values *U_B*, calls *LW_Flux* with the first argument given by the direct sum of U and *U_B*. Finally, U is updated in place using (4).

In finite difference calculations of solutions to hyperbolic conservation laws, it is often the case that the accuracy of the computed solution for a given rectangular mesh spacing can vary substantially as a function of space and time. If one wants to maintain a uniform level of accuracy in a calculation, it is necessary to to vary the mesh spacing as a function of space and time, concentrating computational effort in regions where the error is largest. One approach is Adaptive Mesh Refinement (AMR) [3,2], in which the finite difference mesh is locally refined in response to some locally computed measure of the error. This leads to an algorithm in which the solution is defined on a hierarchy of rectangular grids, with the time evolution computed by multiple applications of a rectangular grid integration scheme such as the Lax-Wendroff algorithm described above. In addition, the error is periodically measured, and the grid hierarchy modified as required. In the following, we describe in detail the structure of the solution on the grid hierarchy, and give a FIDIL implementation of

```
#include "operator_calculus.h"
let
    nvar   = 4,
    Vector = [1 .. nvar] real,
    Values = [*2] Vector,
    Fluxes = [1 .. 2] flex Values,
    gamma = 1.4;
let LW_Flux = proc(Values U; real h,dt) -> Fluxes:
  begin
    let
        U_corner = 0.25*(E1(1) + Id + E2(1) + E1(1)#E2(1))(U),
        F_x = 0.5*((E2(1) + Id)#(E1(1) - Id))(F_fcn@(U)),
        G_y = 0.5*((E1(1) + Id)#(E2(1) - Id))(G_fcn@(U)),
        U_half = U_corner - dt*(F_x + G_y)/(2.*h);

    return [0.5*(Id + E2(1))(F_fcn@(U_half)),
            0.5*(Id + E1(1))(G_fcn@(U_half))];
  end;
let LW = proc(ref Values U; real h,dt):
  begin
    let
        D = boundary(domainOf(U)),
        U_B = phi(U, D, h),
        Flux = LW_Flux(U (+) U_B, h, dt);
    U := U + dt*Div(Flux)/h;
  end;
let F_fcn = proc(Vector U) -> Vector:
  begin
    let p = (U[4] - (U[2]**2 + U[3]**2)/(2.*U[1]))*(gamma - 1.);
    return([U[2],U[2]**2/U[1] + p,U[2]*U[3]/U[1],U[2]*(U[4] + p)/U[1]]);
  end;
let G_fcn = proc(Vector U) -> Vector:
  begin
    let p = (U[4] - (U[2]**2 + U[3]**2)/(2.*U[1]))*(gamma - 1.);
    return([U[3],U[2]*U[3]/U[1],U[2]**2/U[1] + p,U[2]*(U[4] + p)/U[1]]);
  end;
```

FIG. 5. *FIDIL program for the Laz-Wendroff algorithm.*

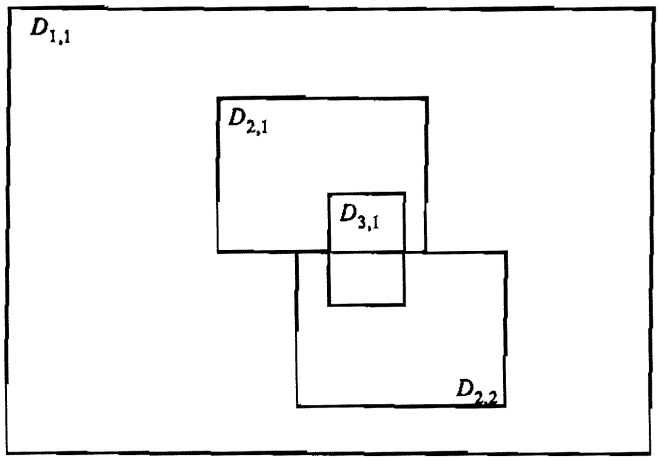


FIG. 6. *Grid hierarchy for three levels.*

the time evolution of that solution for the special case of the Lax-Wendroff algorithm as the underlying integration scheme.

AMR is based on using a sequence of nested, logically rectangular meshes on which the PDE is discretized. For simplicity, we will also require that all the meshes be physically rectangular, with equal mesh spacing in both coordinate directions. We say a mesh at level l is a grid $D_{l,k}$ with mesh spacing h_l and define

$$D_l = \cup_k D_{l,k}.$$

The mesh spacings on the various grids are related by $h_l/h_{l+1} = r$ where the refinement ratio r is restricted to be an even number. By identifying a grid with the domain it covers, we have $D_1 = \cup_k D_{1,k} = D$, the problem domain. If there are several grids at level 1, the grid lines must align with each other, that is, each grid is a subset of a rectangular discretization of the whole space. We may often have overlapping grids at the same level, so that $D_{l,j} \cap D_{l,j'} \neq 0$, but how the grids intersect should have no effect on the solution. We require that the discrete solution be independent of how D_l is decomposed into rectangles. Grids at different levels in the grid hierarchy must be "properly nested." This means

(i) a fine grid is anchored at the corner of a cell in the next coarser grid.

(ii) There must be at least one level $l-1$ cell in some level $l-1$ grid separating a grid cell at level l from a cell at level $l-2$, unless the cell abuts the physical boundary of the domain.

Note that this is not as strong a requirement as having a fine grid contained in only one coarser level grid.

Grids will be refined in time as well as space, by the same mesh refinement ratio. Thus,

$$\frac{\Delta t_l}{h_l} = \frac{\Delta t_{l-1}}{h_{l-1}} = \cdots = \frac{\Delta t_1}{h_1}$$

and so the same difference scheme is stable on all grids. This means more time steps are taken on the finer grids than on the coarser grids. This is needed for increased

accuracy in time. In addition, the smaller time step of the fine grid is not imposed globally. Finally, $U^{l,k}$ denotes the solution on $D_{l,k}$; in addition, we can define U^l to be the solution on D_l, since the solution on overlapping grids at the same level are identical.

The AMR algorithm for advancing the solution on the composite grid hierarchy described above can be formulated as being recursive in the level of refinement. On a given level of refinement l, the algorithm can be broken up into three steps.

Step 1.. Advance the solution on all the level l grids by one time step, using a conservative algorithm for doing so on a single rectangular grid. The only difficulty is in specifying the values along the outer border of $D_{l,k}$. For cells in that border contained in other grids at the same level, we copy the values from those other grids. For cells exterior to the physical domain, we use an appropriate variation of the physical boundary condition operator ϕ. For any remaining cells, we use values interpolated from the coarser levels. For the Lax-Wendroff algorithm described above, we can use a particularly simple interpolation scheme consisting of piecewise constant interpolation in space and linear interpolation in time using only the level $l-1$ grids. This is possible due to the fact that Lax-Wendroff requires a border of boundary values that is only one cell thick, and because of the proper nesting requirement of the AMR grid hierarchy. After the solution is advanced, we use the numerical fluxes to initialize or update certain auxiliary variables used to maintain conservation form at the boundaries between coarse and fine grids; these quantities will be described in detail in step 3.

Step 2.. Advance the solution on all the level $l+1$ grids by r time steps, so that the latest values of the $l+1$ are known at the same time as the level l solutions obtained in step 1.

Step 3.. Modify the solution values obtained in step 1 to be consistent with the level $l+1$ fine grid solutions. This will be the case if

(i) the grid point is underneath a finer level grid;

(ii) the grid point abuts a fine grid boundary but is not itself covered by any fine grid.

In case (i), the coarse grid value at level $l-1$ is defined to be the conservative average of the fine grid values at level l that make up the coarse cell. After every coarse integration step, the coarse grid value is simply replaced by this conservative average, and the value originally calculated using (4) is thrown out. For a refinement ratio of r, we define

$$U^l_{i,j} := \frac{1}{r^2}\sum_{p=0}^{r-1}\sum_{q=0}^{r-1} U^{l+1}_{k+p,m+q}, \tag{6}$$

where the indices refer to the example in Figure 7.

In case (ii), the difference scheme (4) applied to the coarse cell must be modified. According to (4), the fine grid abutting the coarse cell has no effect. However, for

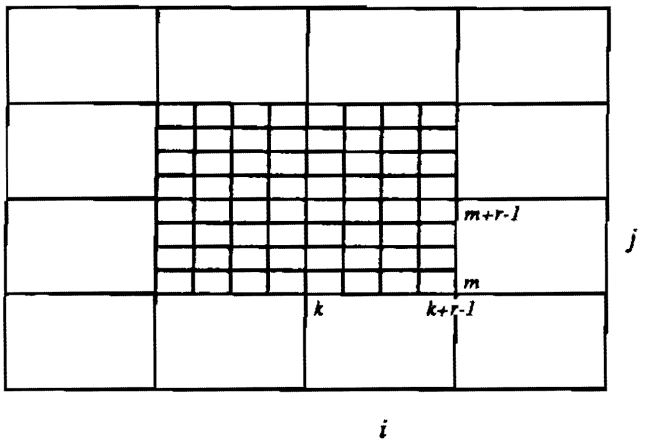


FIG. 7. *The coarse cell value is replaced by the average of all the fine grid points in that cell.*

the difference scheme to be conservative on this grid hierarchy, the fluxes into the fine grid across a coarse cell boundary must equal the flux out of the coarse cell. We use this to redefine the coarse grid flux in case (ii). For example, in the figure below, the difference scheme at cell (i,j) should be

$$
\begin{aligned}
U^l_{i,j}(t+\Delta t_l) &= U^l_{i,j}(t) \\
&+ \frac{\Delta t_l}{h_l}[F^l_{i+1/2,j}(t) - \frac{1}{r^2}\sum_{q=0}^{r-1}\sum_{p=0}^{r-1} F^{l+1}_{k+1/2,m+p}(t+q\Delta t_{l+1})] \qquad (7) \\
&+ \frac{\Delta t_l}{h_l}[G^l_{i,j+1/2}(t) - G^l_{i,j-1/2}(t)].
\end{aligned}
$$

The double sum is due to the refinement in time: for a refinement ratio of r, there are r times as many steps taken on the fine grid as the coarse grid. If the cell to the north of (i,j) were also refined, the flux $G^l_{i,j+1/2}$ would be replaced by the sum of fine fluxes as well.

This modification is implemented as a correction pass applied after a grid has been integrated using scheme (4), and after the finer level grids have also been integrated, so that the fine fluxes in (7) are known. The modification consists of subtracting the provisional coarse flux used in (4) from the solution $U^l_{i,j}(t+\Delta t_l)$, and adding in the fine fluxes to according to (7). To implement this modification, we save a variable δF of fluxes at coarse grid edges corresponding to the outer boundary of each fine grid. After each set of coarse grid fluxes has been calculated in step 1, we initialize any appropriate entries of δF with

$$
\delta F_{i+1/2,j} = -F^l_{i+1/2,j}. \qquad (8)
$$

Since several coarse grids may overlap, it is possible that δF may be initialized more than once. However, since the coarse fluxes on overlapping cell edges for the same level are identical, the initial value so obtained is independent of which particular coarse grid flux is assigned last. At the end of each fine grid time step, we add to

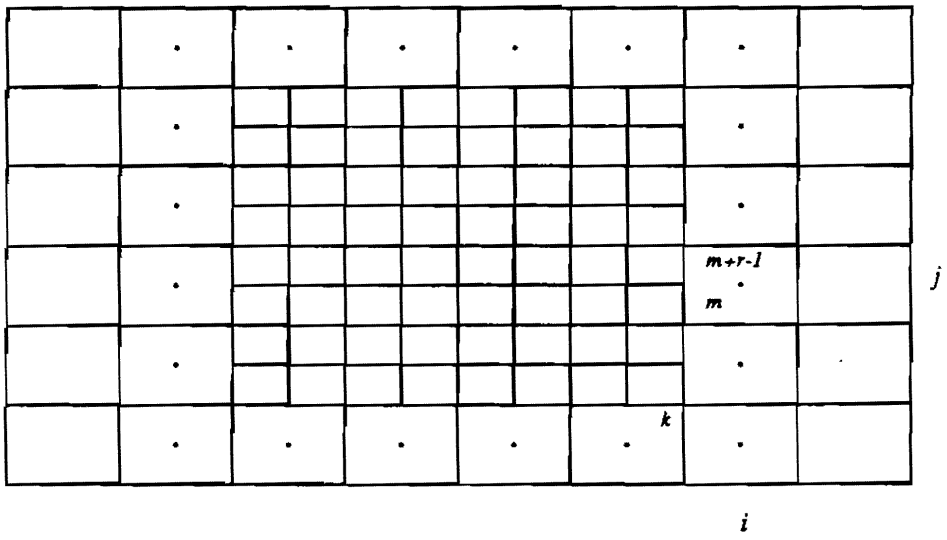


FIG. 8. *The difference scheme is modified at a coarse cell abutting a fine grid.*

$\delta F_{i+1/2,j}$ the sum of the fine grid fluxes along the $(i+1/2,j)^{th}$ edge,

$$\delta F_{i+1/2,j} = \delta F_{i+1/2,j} + \frac{1}{r^2}\sum_{q=0}^{r-1} F^{l+1}_{k+1/2,m+q} \tag{9}$$

Finally, after r fine grid time steps have been completed, we use $\delta F_{i+1/2,j}$ to correct the coarse grid solution so that the effective flux is that of (7). For example, for cell $(i+1,j)$, we make the correction

$$U^l_{i+1,j} := U^l_{i+1,j} + \frac{\Delta t_l}{h_l}\delta F_{i+1/2,j}. \tag{10}$$

If the cell $i+2,j$ were refined, we would also make the correction

$$U^l_{i+1,j} := U^l_{i+1,j} - \frac{\Delta t_l}{h_l}\delta F_{i+3/2,j},$$

and similarly for the vertical fluxes. At the end of a time step, we may have several fine grids available to update a given coarse cell edge, since overlapping grids are permitted. For this reason, one must keep track of the edges of a coarse cell that have already been updated, and only perform the update once for each edge. As before, it doesn't matter which fine grid actually performs the update for any given edge, so the result is independent of the order in which the fine grids are traversed.

In figure 9, we give a FIDIL implementation of the integration step outlined above for AMR. The main procedure *Step* takes a single integer argument l, the grid level being integrated. The principal variables are two copies *U*, *U_new*, of the composite map structure containing the entire set of solution values, with $U^{l,k}$ stored in *U* [*l*] [*k*]. The two sets of values *U*, *U_new* correspond to two different time levels, with times *time* [*l*], *time* [*l*] + *delta_t* [*l*] depending on the grid level *l*. We also define

Boundary_Flux, in which the correction fluxes δF, δG, are stored for the outer edges of all the grids. The refinement ratio s denoted by *nrefine*.

The first step of the AMR integration procedure is performed in (A)–(B). For each level l grid solution $U^{l,k}$, appropriate boundary conditions are calculated and stored in *U_B*, a map whose domain is *Boundary*(*domainOf*(*U* [*l*] [*k*]). In the first loop of this section, boundary values are interpolated from all possible $l-1$ grids, using piecewise constant interpolation in space, and linear interpolation in time. Since the domain of *U_B* is only one cell wide, proper nesting guarantees that all the values of *U_B* corresponding to points in the problem domain D will be set by this procedure. In the second loop, all of the values of *U_B* for which level l values are available are overwritten with them. Then the physical boundary condition procedure ϕ is called to fill in any remaining cells which extend outside D. Having obtained appropriate boundary values for $U^{l,k}$, we compute fluxes using *LW_Flux* and compute *U_new* using (4). Finally, we set *Boundary_Flux* [*l*] and *Boundary_Flux* [*l*+1] using (8) and (9) along the outer edges of $D_{l,k}$. The procedure *Project_Flux*, defined at the end of the figure, calculates the average of the fluxes in the right hand side of (9).

The second step is performed at (C), with *Step* called recursively *nrefine* times with argument $l+1$.

The third step is in (D)–(E). In the first part of the loop over *Grids*, level l cells are incremented using the refluxing algorithm (10). The domains *D_fix* are used to keep track of which edges are being updated, so that no edge gets updated more than once. The final loop over *Gridsp* overwrites the level l values with the averages of the level $l+1$ values using (6).

```
export Step;

let
   maxlev  = 3,
   nrefine = 4,
   refine  = [nrefine,nrefine],
   Box     = [0 .. nrefine - 1, 0 .. nrefine - 1];

let
   Levels  = [1 .. maxlev];

let e_vector =
      proc (integer i, ndim) -> [1 .. ndim] integer:
         [j from [1 .. ndim] : if i = j then 1 else 0 fi ];

postfix operator ^*;
let
   ^* =
      proc(domain[?ndim] D) -> [1 .. ?ndim] domain[?ndim]:
         [i from [1 .. ndim] : D + D << e_vector(i, ndim) ],
   del =
      proc(domain[?ndim] D) -> [1 .. ndim] domain[ndim]:
         [i from [1 .. ndim] :
            (D + D << e_vector(i,ndim)) - (D - D << e_vector(i,ndim))];


external [Levels] flex [] flex Values  U, U_new;
external [Levels] flex []  Fluxes      Boundary_Flux;
external [Levels] real                 delta_t, h, time;
```

FIG. 9. *FIDIL program for Adaptive Mesh Refinement (Part 1 of 4).*

```
let Step = proc (integer l) :
begin                                                    /* (A) */
   U[l] := U_new[l];
   alpha = (time[l-1] - time[l])/delta_t[l-1]

   let
      Grids = domainOf(U[l]),
      Gridsp = if l < maxlev then domainOf(U[l+1]) else NoGrids fi,
      Gridsm = if l > 1 then domainOf(U[l-1]) else NoGrids fi;

   for k from Grids do
      [Boundary(domainOf(U[l][k]))] Vector U_B;

      for km from Gridsm do
         for i from Box do
            contract(U_B << i, refine) *:=
               alpha*U[l-1][km] + (1. - alpha)*U_new[l-1][km]
         od;
      od;

      for kb from Grids do U_B *:= U[l][kb] od;

      U_B *:= phi(U,domainOf(U_B),h[l]);

      let
         Flux = LW_Flux(U[l][k] (+) U_B, delta_t[l], h[l]);

      U_new[l][k] := U[l][k] + delta_t[l]*Div(Flux)/h[l];
      if l > 1 then
         Boundary_Flux[l][k] *:= Boundary_Flux[l][k] + Project_Flux(Flux);
      fi;
      for kp from Gridsp do
         Boundary_Flux[l+1][kp] *:= Flux*nrefine**2
            on del(contract(U[l+1][kp], refine))
      od;
   od;
```

FIG. 9. *FIDIL program for Adaptive Mesh Refinement (Part 2 of 4).*

```
time[l] := time[l] + delta_t[l];                                    /* (B) */

if l < maxlev then

   for n from [1 .. nrefine] do Step(l+1) od;                       /* (C) */

   for kp from Gridsp do                                            /* (D) */
      Boundary_Flux[l+1][kp] :=
         delta_t[l]*Boundary_Flux[l+1][kp]/(h*nrefine**2)
   od;

   for k from Grids do

      [1 .. 2] domain[2] D_fix;

      D_fix := domainOf(U[l][k])^*;

      for dir from [1 .. 2] do

         let E = e_vector(dir, 2);

         for kp from Gridsp do
            U_new[l][k] *:=
               U_new[l][k]
               + (Boundary_Flux[l+1][kp][dir] on D_fix[dir]);
            U_new[l][k] << -E *:=
               U_new[l][k] << -E
               - (Boundary_Flux[l+1][kp][dir] on D_fix[dir]);
            D_fix[dir] :=
               D_fix[dir] -domainOf(Boundary_Flux[l+1][kp][dir])
         od;

      od;
```

FIG. 9. *FIDIL program for Adaptive Mesh Refinement (Part 3 of 4).*

```
            for kp from Gridsp do

               U_new[l][k] *:= [contract(U_new[l+1][kp], refine): 0.0];

               for i from Box do
                  U_new[l][k] *:=
                     U_new[l][k]
                     + contract(U_new[l+1][kp] << i, refine)
               od;

            od;
         od;
      fi;
   end /* of Step */;                                          /* (E) */

   let Project_Flux = proc(Fluxes Flux) -> Fluxes:
   begin

      Fluxes OutFlux;

      domainOf@(OutFlux) := contract@(Flux, refine);

      for dir from [1 .. 2] do

         OutFlux[dir] := [domainOf(OutFlux[dir]): [ [1 .. nvar]: 0.0 ] ];
         let E = e_vector(if dir = 1 then 2 else 1 fi, 2);
         for i from [0 .. nrefine - 1] do
            OutFlux[dir] := OutFlux[dir] + contract(Flux[dir] << (i*E),refine) ;
         od;

      od;
      return(OutFlux);
   end /* of Project_Flux */;
```

FIG. 9. *FIDIL program for Adaptive Mesh Refinement (Part 4 of 4).*

Acknowledgement. We would like to thank Luigi Semenzato for his comments on various versions of this document.

REFERENCES

[1] C. R. ANDERSON, *A method of local corrections for computing the velocity field due to a distribution of vortex blobs*, J. Comput. Phys., 62 (1986), pp. 111–123.

[2] M. J. BERGER AND P. COLELLA, *Local adaptive mesh refinement for shock hydrodynamics*, Tech. Rep. UCRL-97196, Lawrence Livermore National Laboratory, 1987. To appear in *J. Comput. Phys.*

[3] M. J. BERGER AND J. OLIGER, *Adaptive mesh refinement for hyperbolic partial differential equations*, J. Comput. Phys., 53 (1984), pp. 484–512.

[4] A. J. CHORIN, *Numerical study of slightly viscous flow*, J. Fluid Mech., 57 (1973), pp. 785–796.

[5] O. HALD, *Convergence of vortex methods, II*, SIAM J. Numer. Anal., 16 (1979), pp. 726–755.

[6] R. D. RICHTMYER AND K. W. MORTON, *Difference Methods for Initial Value Problems*, Interscience, New York, 1967.