Intelligent Automated Grid Generation for Numerical Simulations

Ke-Thia Yao kyao@cs.rutgers.edu
Andrew Gelsey gelsey@cs.rutgers.edu
Computer Science Department
Rutgers University
New Brunswick, NJ 08903, USA

Abstract

Numerical simulation of partial differential equations (PDEs) plays a crucial role in predicting the behavior of physical systems and in modern engineering design. However, in order to produce reliable results with a PDE simulator, a human expert must typically expend considerable time and effort in setting up the simulation. Most of this effort is spent in generating the grid, the discretization of the spatial domain which the PDE simulator requires as input. To properly design a grid, the gridder must not only consider the characteristics of the spatial domain, but also the physics of the situation and the peculiarities of the numerical simulator. This article describes an intelligent gridder that is capable of analyzing the topology of the spatial domain and of predicting approximate physical behaviors based on the geometry of the spatial domain to automatically generate grids for computational fluid dynamics simulators. Typically gridding programs are given a *partitioning* of the spatial domain to assist the gridder. Our gridder is capable of performing this partitioning. This enables the gridder to automatically grid spatial domains with a wide range of configurations.

Copy-edited version to appear in *AI EDAM (Artificial Intelligence for Engineering Design, Analysis and Manufacturing).*

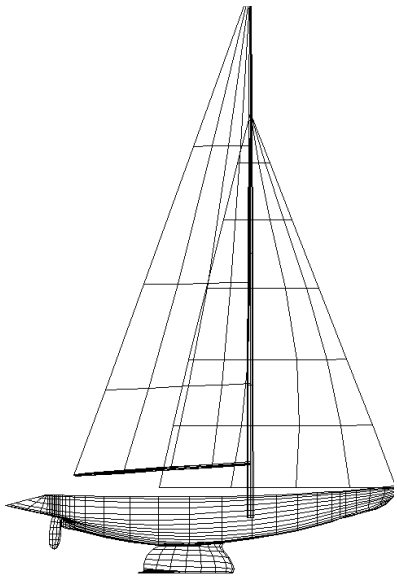

Figure 1: *Stars & Stripes*, winner of the 1987 America's Cup competition

1 Introduction

Numerical simulation of physical systems plays a crucial role in engineering design. Unfortunately, getting simulation results with acceptable accuracy is a time-consuming and labor-intensive process. Although the amount of computational time needed to execute the numerical simulation is considerable, it may not be the dominant factor. In simulations that apply PDE solvers to physical systems with complicated geometries, the most time-consuming portions are rather setting up the numerical simulation, verifying the correctness of the simulation results, and modifying the setup if the results are not within expected tolerances.

Partial differential equation solvers require a grid, a discretization of the spatial regions of interest. Usually in computational fluid dynamics, the spatial regions of interest are either the volumes of space that contain the fluid, or the wetted surface, the surface areas that the fluid contacts. The first type of spatial region requires *volume grids*, and the second requires *surface grids*. Currently, our gridder is limited to generating surface grids. The quality of the grid strongly affects the accuracy and the convergence

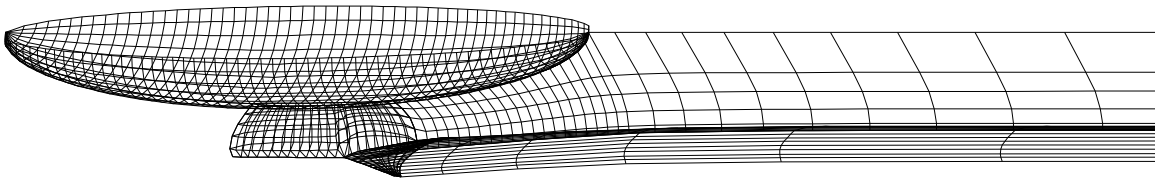

Figure 2: Yacht below waterline with wake sheets. It consists of three components: hull, keel, and winglet

properties of the resulting simulation. Generating a proper grid involves reasoning about the geometry of the regions of interest, the physics of the situation and the peculiarities of the numerical solver.

To deal with the complexities of gridding, the current trend in the field is toward interactive gridding, see Remotique *et al.* (1992), and Kao and Su (1992). The interactivity more readily taps into the knowledge and visuospatial reasoning abilities of the human user through the use of a graphical interface. Interactive gridding enables the user to see the surfaces to be gridded on the computer screen. Rotating and scaling features allow the user to get a better understanding of the surfaces. With this understanding the user is able to visualize the fluid flow over the surfaces, and identify important features of the flow. Then, with the mouse the user is able to interactively construct a grid that conforms to the physical features. Physics conforming grids have good numerical convergence properties. Typically, they provide physically realistic results, and they take less time and space to simulate.

However, this approach is not acceptable for automated design systems, such as the Design Associate (DA), see Ellman *et al.* (1992), for racing yachts like the one in Figure 1. In the process of designing a yacht, the DA must repeatedly evaluate candidate yacht designs. A large number of these evaluations are required, so the capability to automatically evaluate the performance of a candidate yacht design without human intervention is crucial for the success of the DA.

We are working in the physical domain of fluid dynamics, in particular potential flows modeled by Laplace's partial differential equation:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} = 0, \tag{1}$$

where is u is the velocity potential function. The gradient of u, ∇u, is the velocity vector function. The potential flow solver we use is PMARC, a product of NASA Ames Research Center, see Hess (1990), and Katz and Plotkin (1991). The input PMARC requires is a panelization — a discretization of an object's wetted surface as a grid of *surface patches*, where each surface patch is an array of approximately planar quadrilateral panels. This array of panels is represented in PMARC as a matrix of corner points. See Figure 2 for a grid of a yacht automatically generated for PMARC by our gridding program.

The yacht in Figure 2 consists of three input components: an ellipsoid hull, the *Star & Stripes* keel, and the *Star & Stripes* winglet.[1] The wake sheets attached to the rear of the yacht are the vortices shed by the yacht. Discussions on how to attach wakes and how to determine the shape of the wakes are beyond the scope of this article. See Katz and Plotkin (1991) for more discussion of wakes. The *Star & Stripes* winglet attached to the bottom of the keel is considered a major innovation in the field of racing yachts, and the success of the *Star & Stripes* was in part due to its winglet. Current automated gridding programs should be, but are not, able to handle this kind of innovative *topological* change in design without human assistance. In this article we describe an automated gridder that is capable of gridding geometries on wide range of topological configurations.

The input to the gridder is expressed in a language we have developed called Boundary Surface Representation (BSR). Figure 6 graphically depicts the BSR input for this yacht example. This yacht example is used throughout this article. Both BSR and the input will be discussed in much more detail later. For now we point out that BSR input consists of two major parts: *geometrical* and *topological*. The geometrical part represents the detailed features of the yacht, which are the three surface mappings (`SHAPE`) in the figure. The *topological* part represents the surface patches and their adjacency information, which is represented by dotted lines in the figure.

The gridder uses a model-based approach to transform the input to the output. If a grid is to provide satisfactory results, it must conform to the physics of the flow. In potential flow the physics of the flow is determined by the velocity potential function, which actually is the output of the CFD code. To break this cycle we use a simplified model to approximate physics

[1]The *Star & Stripes* is the yacht that won the 1987 America's Cup Competition.

1. Partition each surface into surface patches
2. Parametrize each surface patch
3. Distribute grid lines over each surface patch

Figure 3: Gridding top-level algorithm

of the flow. Then, how the approximate model interacts with the geometry is examined. Visuo-spatial techniques are used to extract relevant physical interaction features. Finally, these features are used to aid in the gridding process.

This article contributes to model-based reasoning, spatial reasoning, and intelligent scientific computation in the following ways:

- This article demonstrates how simple models may be used to predict the approximate behaviors of a physical system and how qualitative (topological) information may be deduced from the approximate behavior to aid in obtaining more accurate behaviors from complex (PDE) models.
- Imagistic reasoning and visualization play important roles in gridding and more generally in problem solving and in reasoning about physical systems, see Yip (1991), and Narayanan (1993). In the gridding domain this article presents a program that exhibits these abilities by making use of physical domain knowledge and geometric knowledge.
- The gridder in this article may be classified as a second generation AI gridding system as defined by Andrews (1987). It is capable of automated construction and synthesis of a grid, instead of classifying and selecting from pre-enumerated solutions.

2 Why is automated gridding hard?

2.1 Steps to gridding

We divide gridding into three steps as shown in Figure 3. The first step is to *partition* the input surface into griddable *surface patches*. This step finds the appropriate boundary lines (or *partitioning lines*) for the surface patches. This step is often the most difficult to automate, because it involves significant physical and geometrical reasoning.

Step two, for each surface patch, *parametrize* it by defining two families of approximately orthogonal grid lines. A formal definition will be given later when BSR is defined. But intuitively, suppose a surface patch is the xy-plane, then $\{x = constant,\ y = constant\}$ is one possible parametrization, and $\{x + y = constant,\ x - y = constant\}$ is another.

The last step is to determine how many grid lines to lay down on each of the surface patches, and in particular where to lay them down. This step corresponds to picking the constants to instantiate the parametrization equations in step two. The intersections of these grid lines form corner points of the array of panels, which is the input to PMARC. This step we shall call the grid line *distribution* step. The distribution of grid lines can make grids with the same parametrization look different and may make the numerical simulator behave differently. For example, to grid a line segment from 0 to 1 using the *equal-distance* distribution scheme, $x = i/10$, where $i = 0, \ldots, 10$, may make the numerical simulator converge slower than a *cosine* distribution scheme, $x = (1 - \cos \pi i/10)/2$, where $i = 0, \ldots, 10$.

2.2 Evaluation criteria

Gridding as defined by Figure 3 is unconstrained. The ultimate test for a grid is to check how sound the resulting simulation is, and how well it resolves the physical features of the domain. Short of feeding the grid to a simulator, there are ways of checking the goodness of a grid.

Through our discussion with hydrodynamicists we have formulated a list of grid evaluation criteria and constraints. On the basis of the geometric properties of the grid, these evaluation criteria attempt to predict the soundness of PMARC's output. We divide this list into four levels, ranging from constraints that absolutely must be satisfied to heuristic advice based on

1. Simple connectedness constraint: surface patches must be simply connected, i.e., no holes.
2. Coverage constraint: patches must not overlap or leave gaps.
3. Planarity constraint: panels must be approximately planar.
4. Heuristic criteria:
    - follow-streamlines: grid lines should follow the streamlines of the fluid flowing over the body.
    - expansion ratio: the area of the adjacent panels should not increase by more than a fixed ratio.
    - orthogonality: grid lines should intersect at right angles.

Figure 4: Grid evaluation criteria

experiences of our experts. See Figure 4.

For a CFD simulator as complicated as PMARC it is difficult to derive a complete numerical error model. Typically fragments of the simulator are analyzed to check what implications they have on gridding. The reasoning behind these criteria can be traced back to numerical properties of PMARC. Thus, instead of having a complete model of the simulator, we have a set of numerical model fragments.

The level one *simple connectedness* constraint is actually a statement about the geometric representation used by PMARC. PMARC represents surface patches as a matrix of corner points of adjacent panels. This type of representation does not allow for holes between panels. Panelization of surfaces with holes must be done with multiple patches.

The level two *coverage constraint* can be considered as a problem independent statement regarding the correctness of a grid. If a grid is to be correct for any physical situation at all, the adjacent surface patches of the grid must meet along a curve. They cannot overlap nor leave gaps. They must exactly cover the wetted surface.

A problem dependent statement of the correctness of grids states that grids must be faithful discretizations of the actual surface. One implication of problem dependent correctness is the *planarity* constraint. Each panel is

represented by the points at its four corners, which may or may not be coplanar. If the corner points are not coplanar they cannot completely represent a body's actual surface.

PMARC associates a constant number, velocity potential, with each panel in the grid. In order to compute the velocity vector PMARC needs to take the gradient of these velocity potentials. For each panel PMARC calculates its gradient by applying central differencing using its four adjacent panels. The truncation error caused by non-uniform spacing is directly related to the rate of change of grid spacing. This leads to the *expansion ratio* criteria. Similarly, numerical error tends to be minimized if the adjacent panels are *orthogonal* with respect to each other. Assuming the surface of interest is approximately planar, [Thompson *et al.* 1985] is able to show that the non-uniform spacing term of the truncation error is proportional to second derivative of the grid spacing function times the second derivative of the velocity potential. Under further assumptions Thompson is able to show that the truncation error varies inversely with the sine of the angle between the grid lines. See [Thompson *et al.* 1985] for more details.

A *streamline* is a line traced out by following the flow velocity vectors. For any point on a streamline its tangent is the flow velocity vector, ∇u. The *follow-streamline* criteria is derivable from numerical accuracy and computational efficiency considerations. If this criterion is satisfied, fewer panels are needed for the same numerical accuracy. Consider a simple case were the flow is restricted to the xy-plane and the velocity potential is $u(x, y) = cx$, i.e., the flow, $\nabla u = (c, 0)$, is flowing uniformly in the x-direction. In this case the streamlines are $x = constant$ lines. Restricting the error in potential to be less than or equal to some parameter h, then $\{x = 2ih, y = j\}$, where i and j are integers, is an parametrization/distribution scheme that satisfies the *follow-streamline* criteria. This scheme, $\{x+y = 2ih, x-y = 2jh\}$, places grid lines at a forty-five degree angle to the streamlines. Both divide the xy-plane into panels, where in each panel the difference between the maximum potential and the minimum potential is $2h$. By picking the potential to be the average of the two extremes, the error is exactly h. But to achieve this h error, the *follow-streamline* conforming scheme only needs $1/2h$ panels per unit square, while the other scheme need $1/4h^2$ panels per unit square. The *follow-streamline* scheme is able to get this $O(1/h)$ savings, because it only needs to refine the grid in one direction, the direction of the flow. The other scheme needs to refine the grid in both directions.

These evaluation criteria in their present, qualitative form cannot easily be used to form actual grids. We have quantified these criteria and incorporated them into our program. Given a grid, the program is able to judge how well the grid satisfies the evaluation criteria.

The hydrodynamicists formulated the above evaluation criteria for use with the PMARC simulator, however, the evaluation criteria are quite general. They should apply to other potential flow simulators as well. The *orthogonality* and *expansion ratio* criteria apply to any simulator that uses numerical differentiation. The *follow-streamlines* criterion applies to any potential flow simulator. The *planarity* and *coverage* constraints applies to any grid that properly presents the physical situation. The *simple connectedness* constraint is only simulator dependent constraint, although most simulators require it. Enforcing this constraint for simulators that do not require it results in the creation of addition surface patches. This makes the grid more complicated, but should not affect the solutions returned from the simulators.

2.3 Difficulties of partitioning

Much work has been done on the problem of automated gridding [e.g., see Thompson *et al.* (1985), and Knupp and Steinberg (1993)], and many gridding programs have been developed. However, most of these efforts concentrate on developing new methods of parametrization and new distribution schemes. The choices of which parametrization method and which distribution scheme to use are usually left to the human expert. Almost no work has been done on automated partitioning.

Most of the programs rely exclusively on the human expert to do the partitioning. He is expected to do the partitioning by either writing batch commands, or more recently by using an interactive graphical interface. In either case, the partitions generated only apply to the one particular problem at hand. More recently, Schuster (1992) has been trying to revive batch mode gridding by writing more general batch commands. However, his program is only able to grid a small, fixed set of airplane topologies.

One of the fundamental problems with the current automated gridding programs is that they do not make use of topology. All the topological information has been distilled away by either having the user provide the partitions or by fixing the possible topologies. The programs can only work on individual surface patches. Another problem is that programs have neither

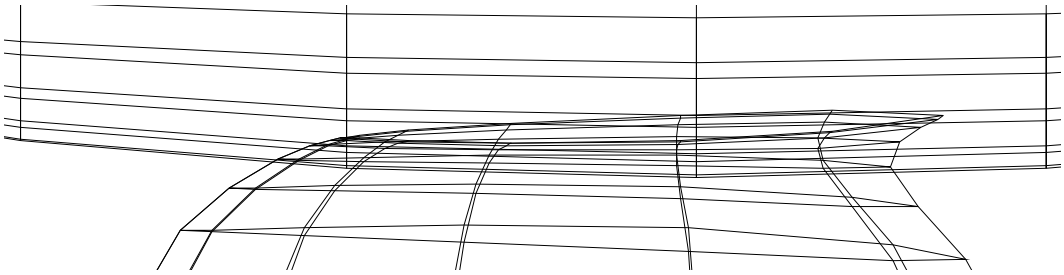

Figure 5: Hull-keel intersection

the knowledge of physics nor the knowledge of numerical analysis needed to generate grids that will lead to good simulations.

One manifestation of the failure to include topological information is shown in Figure 5, which shows how the *coverage* constraint may be violated. A closer examination of the surface area near where the hull and keel meet reveals that the keel actually protrudes into the hull, and the hull has an extra surface area where the keel is. Surfaces given to the gridding program often contain *fictional surface areas*, areas that should not be gridded. Fictional surface areas are useful because they allow the hull and keel to be modified independently while still remaining in contact. However, an automated gridding program must be able to distinguish between the real and fictional areas in order to satisfy the *coverage* constraint.

Recall that PMARC represents each patch by a matrix of corner points. This type of representation does not allow for holes in patches, i.e., the patches must be *simply connected*. If the gridding program has knowledge of the underlying numerical analysis program, it would realize that once it removes the fictional surface area from the hull, it must break the hull in half to "cut" out the hole. This cut can be performed in many ways, but how it is done affects how easily the parametrization and distribution steps can be performed to satisfy the evaluation criteria.

As we shall see the *follow streamline* criterion provides strong guidance on resolving gridding decisions. For example, the *follow streamline* criterion suggests holes should be cut out along streamlines. However, gridding programs typically lack the physical knowledge to make use of this guidance.

In the following section, we first present a geometric language, Boundary Surface Representation (BSR), which is capable of representing geometrical

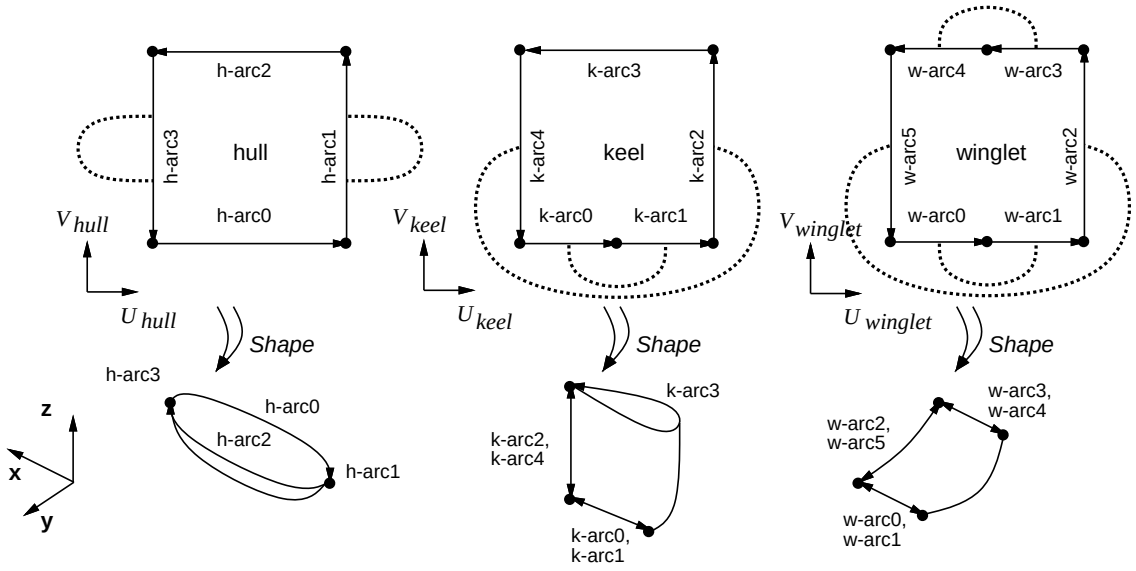

Figure 6: BSR input

information, topological information as well as associating attributes of the physical domain to the geometry. Then, we present a principled method of solving the partitioning, parametrization, and distribution problems for streamlined objects based on reasoning about physics of the flow domain. We call this method Streamline-Based Gridding. Then, in the algorithm section we present detailed algorithms implementing the streamline-based gridding method.

3 Boundary Surface Representation(BSR)

The BSR representation of the yacht example is depicted in Figure 6. The yacht consists of three components: hull, keel, and winglet. Each component is represented by a surface. We shall use this example to help describe BSR.

3.1 Domains and Mappings

In BSR a surface is represented parametrically. A surface is a two dimensional object, so it defines a two-dimensional parametric space. Spaces are called

domains in BSR. For example, the command, `create-domain(hull,2)`, creates a two-dimensional domain, called `hull`. The shape of `hull` is a two-dimensional hyper-cube, $(u_{hull}, v_{hull}) = ([0,\ldots,1],[0,\ldots,1])$.

Surfaces reside in the three dimensional, physical space. In BSR this physical space is a pre-defined domain, called `*XYZ*`. A mapping facility is provided to define the relationship between parametric spaces and `*XYZ*`.

```
hull-shape := create-mapping(ELLIPSOID, major-axis=10,
                             minor-axis=2).
mapping-relation(SHAPE, hull, *XYZ*, hull-shape).
get-mapping(SHAPE,hull,*XYZ*) → hull-shape.
```

The `create-mapping` command creates `hull-shape` ($R^2 \rightarrow R^3$), a mapping that maps from a unit square to an ellipsoid surface in 3D. The `mapping-relation` command specifies that `hull-shape` defines the `SHAPE` of the `hull` domain in `*XYZ*` domain. See Figure 7. Once the `mapping-relation` is executed, `get-mapping` with the appropriate arguments is able to retrieve `hull-shape` mapping. The `create-mapping` command understands several mapping types, such as `ELLIPSOID`, `NACA-WING`, and `BSPLINE`. Also, the user is able create his own mapping by using the mapping type `BLACK-BOX`, and then specifying the function explicitly.

```
hull-normal := create-normal-mapping(hull-shape).
mapping-relation(NORMAL, hull, *XYZ*, hull-normal).
```

The mapping facility is not limited to defining shapes. Other geometric and physical relationships may also be defined. The above two commands define the geometric `NORMAL` relationship for `hull` domain. They define the `hull-normal` mapping ($R^2 \rightarrow R^3$), which specifies the outside normal vector for each point on the ellipsoid hull.

```
hull-flow := create-flow-mapping(free-stream-vector,hull-shape,
                                 hull-normal).
mapping-relation(FLOW, hull, *XYZ*, hull-flow).
```

Using `SHAPE` and `NORMAL` mappings, the command `create-flow-mapping` projects the `free-stream-vector` onto the ellipsoid hull. For each point on

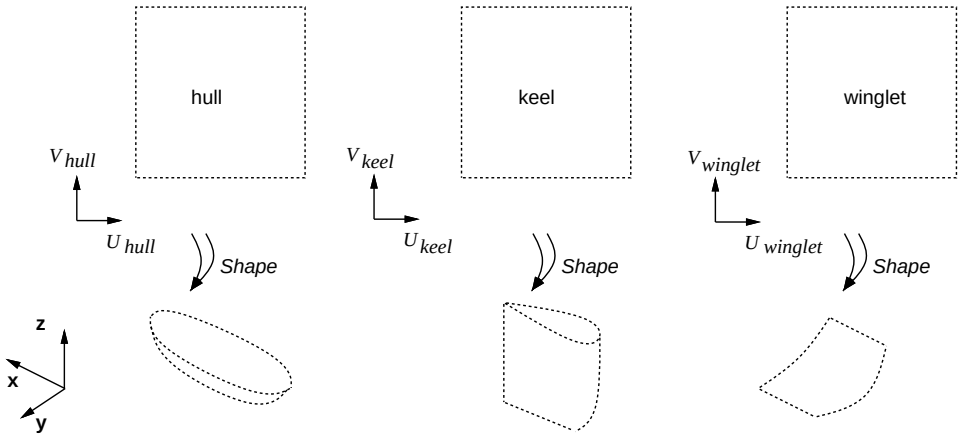

Figure 7: Parametric spaces and `SHAPE` mappings to three dimensional space, `*XYZ*`

the hull, the `SHAPE` and `NORMAL` mappings define a tangential plane. The `FLOW` mapping ($R^2 \rightarrow R^3$) defines a vector on the tangential plane with the smallest angle to the `free-stream-vector`. The `free-stream-vector` is the direction the water would have flowed, if the yacht were not present. This mapping approximates the stream vector for each point on the surface of the elliptic hull.

```
create-domain(keel,2).
keel-shape := create-mapping(BSPLINE, bspline-file="ss87.keel").
mapping-relation(SHAPE, keel, *XYZ*, keel-shape).
```

The domains and mappings for the keel and winglet surfaces are defined similarly. The above three commands show how the `keel` domain and its `SHAPE` mappings are defined. See Figure 7.

3.2 Surface patches and lower dimensional structures

In gridding the ability to represent partial surface areas, *surface patches*, is important. BSR accomplishes this by using an explicit boundary representation, BR, data structure. The boundary of a surface patch is explicitly represented as *arcs* in parametric space. In turn, each arc is bounded by *nodes*.

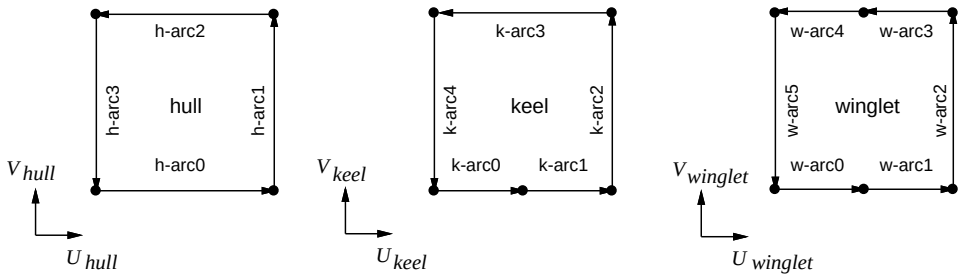

Figure 8: BR representation of parametric space shape

```
domain-shape (hull) → hull-patch.
domain(hull-patch) → hull.
boundary(hull-patch) → ((h-arc0, h-arc1, h-arc2, h-arc3)).
boundary(h-arc0-br) → ((h-node00,h-node01)).
boundary(h-node00-br) → (0,0).
boundary(h-node01-br) → (1,0).
```

In fact the shape of the `hull` parametric space, $(u_{hull}, v_{hull}) = ([0, \ldots, 1], [0, \ldots, 1])$, is represented using BR. See Figure 8. The `domain-shape` command on the domain `hull` returns the BR structure `hull-patch` representing the shape of the domain. Inversely, the `domain` on the BR `hull-patch` returns the domain `hull`. The `boundary` command on `hull-patch` returns a list of lists of BR. Each list of BR is closed sequence of arcs. The first list represents the outer boundary of the surface patch. The remaining lists, if any, represent holes in the interior of the patch. In this case `hull-patch` does not contain any holes. The `boundary` command on an arc returns the two bounding nodes. The `boundary` command on an node returns the coordinates of node in parametric space.

BSR adopts a *counter-clockwise* rule to distinguish between two types of closed sequence of arcs. A counter-clockwise, closed sequence of arcs denotes the area bound by the arcs. A clockwise, closed sequence of arcs denotes the area outside the arcs. This implies the area on the "left-hand side" of an arc is "inside," and the area on the "right-hand side" is "outside." See Figure 9.

```
make-node (h-node00-br,hull,(0,0)).
make-node (h-node01-br,hull,(0,1)).
```

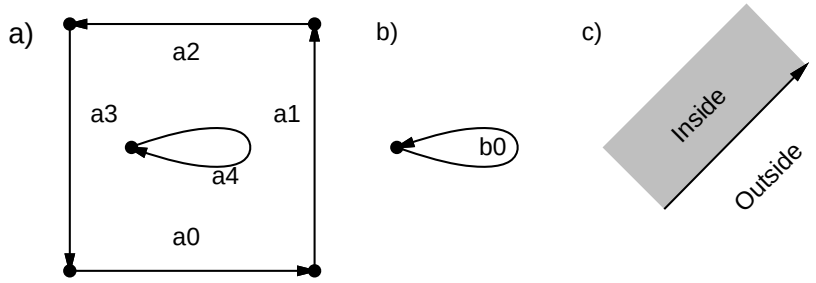

Figure 9: a) Surface patch with a hole. The `boundary` of the patch is `((a0, a1 ,a2, a3), (a4))`. b) Surface patch of the removed area. The `boundary` of the patch is `((b0))`. c) The area on the "left-hand side" of an arc is "inside," and the area on the "right-hand side" of an arc is "outside."

```
make-implicit (h-arc0-br,((h-node00, h-node01))).
...
make-implicit (hull-patch, ((h-arc0, h-arc1, h-arc2, h-arc3))).
```

BSR provides constructor commands to build BR structures. The above show how the hull-patch is constructed. The `make-node` commands create nodes, `h-node00` and `h-node01`, in the `hull` domain with coordinates $(0, 0)$ and $(0, 1)$, respectively. The `make-implicit` takes BR structures in dimension n to generate BR structures in dimension $n+1$. The first `make-implicit` command creates an arc, `h-arc0`, from boundary nodes. The second `make-implicit` command creates a surface patch, `hull-patch`, from boundary arcs.

```
func0(v) := (0,v).
make-explicit (h-arc0-br,((h-node00, h-node01)),
               func0).
```

A surface patch is unambiguously specified by its boundaries, but an arc is *not* unambiguously specified by its boundaries. The interior of an arc must be specified *explicitly*. The reason for the difference is that both `hull` and the surface patch have the same dimension of two, while arc has dimension of one. The `make-implicit` command for arcs is actually a shorthand for the above two commands. In this case the `make-explicit` command uses the function `func0` to determine the interior of the arc. The function `func0` must

satisfy the conditions: `func0(0) = boundary(h-node00)` and `func0(1) = boundary(h-node11)`.

```
create-domain(hull2,2).
func1(u,v) := (u/2,v).
hull2-hull-shape := create-mapping(BLACK-BOX, func0).
mapping-relation(SHAPE, hull2, hull-patch0, hull2-hull-shape).
get-mapping(SHAPE,hull2,hull) → hull2-hull-shape.
get-mapping(SHAPE,hull2,*XYZ*) → hull2-xyz-shape.
```

With BR defined, we can show that mapping from one parametric space to another parametric space is allowed. Suppose surface patch `hull-patch0` is in domain `hull`, and shape of `hull-patch0` a rectangle, $u_{hull} = [0, 0.5]$ and $v_{hull} = [0, 1]$. The first command creates a new domain, `hull2`. The second command defines the function `func0` ($R^2 \rightarrow R^2$) from a unit square to the rectangle. The next command creates a `BLACK-BOX` type mapping, `hull2-hull-shape`. The `mapping-relation` command specifies the `SHAPE` of the `hull2` domain in the `hull` domain is `hull-patch0`. The first `get-mapping` retrieves `hull2-hull-shape`, the `SHAPE` of `hull2` in `hull` domain. The second `get-mapping` retrieves `hull2-xyz-shape`, the `SHAPE` of `hull2` in `*XYZ*` domain. The mapping `hull2-xyz-shape` is constructed automatically by BSR, since BSR knows how to map from `hull2` to `hull`, and then from `hull` to `*XYZ*`.

Notice the boundaries for the keel and winglet surface patches are defined with more arcs than needed to define their shape. The extra arcs are used to express topological information.

3.3 Topological links

Arcs are also useful in expressing topological information. In BSR two arcs are *sewn* together if they are the same line when mapped using `SHAPE` into `*XYZ*`, even though they are distinct in parametric space. Graphically, the sewn relationship is represented as dotted lines, see Figure 10.

```
sew (k-arc2,k-arc4).
sew (k-arc0,k-arc1).
sewn-to (k-arc2) → (k-arc4)
```

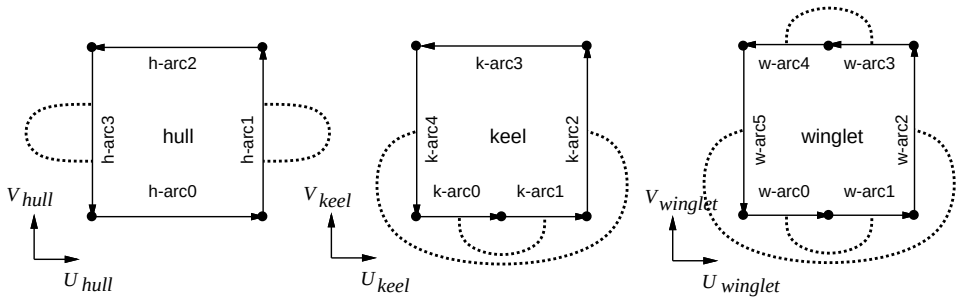

Figure 10: Surface patches with topological links

The first `sew` command sews arc `k-arc2` and arc `k-arc4` together. These arcs map to the trailing edge of the keel. Thus in `*XYZ*` it is possible to travel just in the direction of increasing u_{keel} and end up at your starting point. Together the `sew` commands implies the topology of the keel is similar to that of a cylinder with one end closed, like a "cup."

```
sew (h-arc1,h-arc1).
sew (h-arc3,h-arc3).
```

Sewing an arc to itself is used to show degeneracy. That is, the arc maps to one point in `*XYZ*`. The arc `h-arc3` maps into the trailing point of the hull; the arc `h-arc1` maps into the leading point of the hull.

```
sew (w-arc0,w-arc1).
sew (w-arc3,w-arc4).
sew (w-arc2,w-arc5).
```

Notice each of the arcs on the winglet surface patch is sewn to some other arc on that surface. This means that in `*XYZ*` the winglet surface is a closed surface. A closed surface partitions space into two parts. The winglet surface is the *boundary surface* of the actual, three-dimensional winglet. Of the three surfaces the winglet is the only one that actually encloses some finite volume in `*XYZ*`.

The yacht as represented by the three surfaces does not form a closed surface. An additional *water plane* surface, $z = 0$, is needed to complete the definition of the yacht. However, this plane is not needed by PMARC for the actual simulation. So, BSR does not require the user to input the *water plane*.

```
neighbor (hull-patch) → (hull-patch).
neighbor (h-arc0) → (h-arc1, h-arc2, h-arc3).
neighbor (h-arc1) → (h-arc0, h-arc1, h-arc2).
neighbor (k-arc2) → (k-arc0, k-arc1, k-arc3).
```

With the *sewn* relationship defined, BSR allows *neighbor* relationship queries. The command `neighbor` returns adjacent BR structures of the same dimension. By definition, if an arc is sewn to itself, then it is its own neighbor.

The definition of topology in gridding differs from the standard mathematical definition of topology of surfaces. The topology of a grid is defined in terms of its surface patches and its neighbor relation. Two grids are topological similar if they have the same number of surface patches and the same neighbor relation.

```
create-flow-domain(hull,ELLIPSOID, major-axis=10,
                   minor-axis=2, fsv=free-stream-vector).
create-flow-domain(keel, BSPLINE, bspline-file="ss87.keel",
                   fsv=free-stream-vector).
create-flow-domain(winglet, BSPLINE, bspline-file="ss87.winglet",
                   fsv=free-stream-vector).
```

For the potential flow domain, a special command, `create-flow-domain`, is provided to create all at once: the domain, the `SHAPE` mapping, the `NORMAL` mapping, the `FLOW` mapping, the surface patch, the boundary arcs, and the topological information.

3.4 Gridding steps in BSR

Now, the steps to gridding can be defined more precisely in terms of BSR.

```
partition(domain-1, ..., domain-n) → (patch-BR-1, ..., patch-BR-m).
parametrize(patch-BR-1, ..., patch-BR-m) →
            (param-domain-1, ..., param-domain-m).
distribute(param-domain-1, ..., param-domain-m) → grid-lines.
grid-lines =
```

$$(\{u_{param-domain-1} = (constants),\ v_{param-domain-1} = (constants)\},$$
$$\ldots,$$
$$\{u_{param-domain-m} = (constants),\ v_{param-domain-m} = (constants)\}\ ).$$

The `partition` command takes as input a list of domains, and outputs a list of surface patches in BR representation.

The `parametrize` command takes as input a list of surface patches, and outputs a list of domains. For each input surface patch `patch-BR-j`, there is a corresponding domain `param-domain-j` that maps to that surface patch. That is the mapping retrieved by the command `get-mapping (SHAPE,param-domain-j, domain(patch-BR-j))` specifies that the `SHAPE` of `param-domain-j` in domain `domain(patch-BR-j)` is `patch-BR-j`.

The `distribute` command takes as input as list of domains, and outputs the grid lines in parametric coordinates. For each `param-domain-j` domain a list of constant $u_{param-domain-j}$ grid lines and a list of constant $v_{param-domain-j}$ grid lines are outputed. The intersection grid lines defines the corner points of the panels. The mappings `get-mapping (SHAPE,param-domain-j,*XYZ*)` is used to convert the parametric grid to a three-dimensional Cartesian grid.

4 Streamline-Based Gridding

Streamline-based gridding is based on the following observation. The solution to Laplace's equation depends neither on the current state of the flow nor on time, so the geometry of the object determines the solution. Since streamlines are key characteristics of the solution, analyzing how streamlines interact with geometry provides key insights to qualitative behaviors of Laplace's equation. These insights enable us to determine the topology of streamlines. In turn, this topology provides natural boundaries for patches in grids.

The most immediate problem encountered in streamline-based gridding is how to get the initial set of streamlines. Streamlines are constructed from the velocity vector function, which is the gradient of the potential function. But, during the gridding process PMARC has not yet been run to generate the potential function. Our approach is to use a simplified physical model, called the *projection flow model*. This model is efficient to compute, and it is able to directly approximate the velocity vector function. Our experiments have shown that it captures enough of the physical-geometrical interactions for gridding purposes.

The next step is to study the how the streamlines interact with geometry. First, point features of the interaction of streamlines are classified. Then aggregate features are defined using the point feature. Finally, the streamline

topology are formed from these aggregate features.

4.1 Projection flow model

We have experimented with various methods of predicting the streamlines *a priori*. However, we have found the simple projection of the `free-stream-vector` onto the body surface to be a good approximation of the flow velocity vector. This is the `FLOW` mapping defined earlier. Streamlines are formed by connecting the velocity vectors.

A physical justification for using the *projection flow model* is as follows. This behavior predicted by this model may be viewed as the result of an approximation to a partial solution of Laplace's equation (Eq. 1). Since the right hand side of Eq. 1 is zero, if potential functions u_1 and u_2 are solutions to Eq. 1, then the sum of u_1 and u_2 is also a solution. This is called the superpositioning principle. Using superpositioning PMARC forms the true solution by summing three types of solutions:

$$u = u_{freestream} + \sum_{panel\ i} u_{source\ i} + \sum_{panel\ i} u_{doublet\ i}.$$

The $u_{freestream}$ potential induces the free stream. If the yacht were not present then the $u_{freestream}$ would be the only potential needed. The boundary condition of PMARC states that no flow shall penetrate the surface of the yacht. For each panel i, the $u_{source\ i}$ potential locally induces a flow normal to that panel to cancel out the normal direction effects of the $u_{freestream}$ potential, and prevents the free stream flow from penetrating that panel. But the effect of each $u_{source\ i}$ is felt non-locally by other panels. So correction factors, the $u_{doublet\ i}$ potentials, are needed to cancel out the non-local effects of the $u_{source\ i}$ potentials to satisfy the boundary condition globally. Since $u_{freestream}$ is given and $\sum_{panel\ i} u_{source\ i}$ can be calculated from the geometry, summing these two types of solutions provides a good approximation to the true solution. *Projection flow model* may be viewed as the result of approximating $u_{freestream} + \sum_{panel\ i} u_{source\ i}$, assuming the each panel is actually a point and the effect of $u_{source\ i}$ is felt locally at that point.

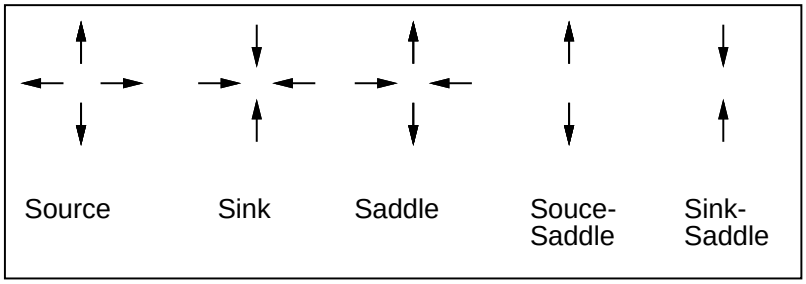

Figure 11: Stagnation Node Classification

4.2 Surface point classification

According to potential flow theory, streamlines on the surface of an object can only start and terminate at *stagnation nodes*, $\nabla u = 0$. At stagnation nodes the fluid stalls either to merge or to divide around the object. Streamline topology surrounding a stagnation node (i.e., the streamline directions of an infinitesimal ϵ-neighborhood of points near the point) in potential flow can be quite complicated with varying arrangements of streamlines entering and leaving. The common types of stagnation nodes are the *source*, *sink*, and *saddle* nodes. At one end of the spectrum is the source node with the all neighboring streamlines flowing away from it. At the other end of the spectrum is the sink node with all streamlines flowing into it. The saddle node is somewhere in between the two extremes with two streamlines flowing away from the saddle in one direction, and two streamlines flowing into it in another direction. Combining the source node with the saddle, it is possible to get the degenerate *source-saddle* node with only two streamlines flowing away from the node and no streamline flowing into it. The *sink-saddle* is defined similarly. See Figure 11.

Points that are not stagnation nodes may be called *unidirectional flow* nodes, or *uniflow* nodes for short. Unlike stagnation nodes, the streamlines in the ϵ-neighborhood of a uniflow node, as well as the streamline at the node itself, flow in the same direction. Typically a surface consists mostly of uniflow nodes with only a small portion of stagnation nodes.

In gridding it is useful to consider point topologies where ϵ is some small, finite number, rather than infinitesimal. We name two such useful topologies as the *flowing-source* node and the *flowing-sink* node. When ϵ approaches

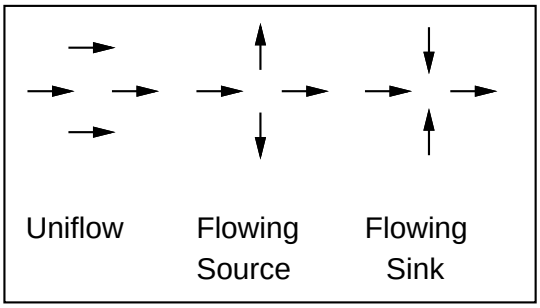

Figure 12: Non-stagnation node Classification

zero both these nodes are uniflow nodes, but with a small ϵ the point topology changes to a combination of uniflow node and source node for the flowing-source, and to a combination of uniflow node and sink node for the flowing-sink node. See Figure 12.

Notice that our stagnation point classification differs from more traditional fluid dynamics classification. We have merged some types of stagnation points and eliminated other types altogether. The reason behind the simplification is that either the finer distinctions are not needed by the gridder, or the types of stagnation point do not occur in the flows we work with. For example, we have eliminated the *center* node. A center node has no streamlines flowing either in or out of it, instead neighboring streamlines form elliptical patterns circling it. This type of stagnation point is not possible with the free stream projection flows. For a more detailed discussion of traditional classification schemes see Tobak and Peake (1982), and Perry and Chong (1987).

4.3 Aggregate features

Surface point classification identifies zero-dimensional features of the flow. We define three types of one-dimensional features: *streamline*, *source line*, and *sink line*.

Streamline is defined earlier. A streamline is a line traced out by following the flow velocity vectors. In terms of point features, it is a sequence of non-stagnation nodes.

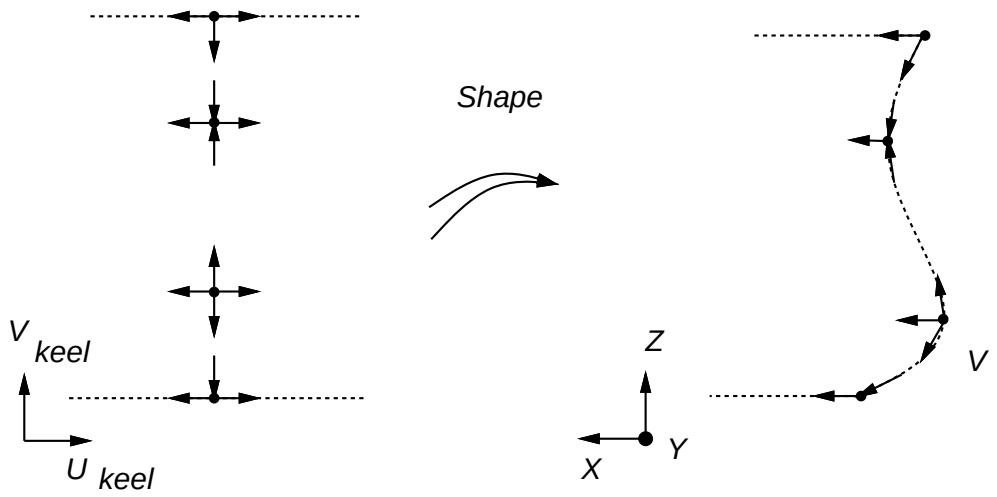

Figure 13: Stagnation nodes on the leading edge of the Stars & Stripes Keel. The right is the keel in Cartesian space, and the left is it in parametric uv space.

Source line is the one-dimensional counterpart of the source node. All neighboring streamlines of a source line flows away from it. There are two types of sources lines. If we require the neighborhood of the source line to be infinitesimal, then the source line is a sequence source-saddle nodes. For example, this type of source line is found on unswept wings, where the leading edge of the wing is orthogonal to the free stream. With a small, finite neighborhood the source line is a sequence consisting of source nodes, saddle nodes, and flowing source nodes. For example, the leading edge of the *Stars & Stripes* keel in Figure 2 consists of a sequence of the following: a *half* source node, a line of flowing-source nodes, a saddle, another line of flowing-source nodes, a source node, one more line of flowing-source nodes, and finally a single *half* saddle node, see Figure 13. Half nodes are nodes found on the boundary of surfaces, so part of the neighboring streamlines are cut off. This pattern of alternating source and saddle nodes, interleaved with flowing-source nodes, must be preserved. *Sink line* is defined similarly.

Even higher dimensional objects can be defined. We define two stream-

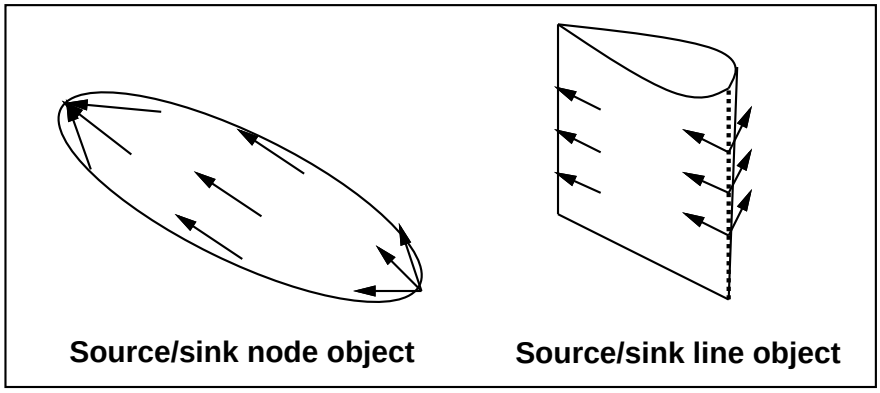

Figure 14: Streamlined object classes

lined object classes as shown in Figure 14. First is the *source/sink node* class. Streamlines on objects from this class all originate from one point on the surface, the source node, and all flow to and terminate at another point on the surface, the sink node. Spheres, ellipsoids and other simple bodies of revolution are objects of this class. These objects have axial symmetry, so there can only be one source node and one sink node.

The second is the *source/sink line* class. This class is like the previous class, except that the streamlines appear to originate and terminate at lines instead of nodes. For instance, the leading edge of a keel is *source line*, and the trailing edge is *sink line*. All the streamlines flow from the leading edge to the trailing edge. Any wing shaped object belongs to this class.

Using only these two object classes, one can already construct complex geometric objects, such as the yacht in Figure 2. The yacht consists of a source/sink node object (hull), and two source/sink line objects (keel and winglet).

4.4 Application to gridding

Based on the *follow-streamline* heuristic for gridding, it is reasonable to grid a source/sink node object as a single surface patch, since all the streamlines are flowing in one direction, from the source node to the sink node. A source/sink

line object should be gridded as two surface patches with the source line and sink line acting as partitioning lines. Although the streamlines still flow from the source line to the sink line, the streamlines take two different routes. For example, in Figure 6 one set of streamlines flows to the sink from the right side of the keel ($u_{keel} > 0.5$), and the other set flows from the left side ($u_{keel} < 0.5$). The source/sink lines separate these two flow regions.

Streamlines are also useful in parametrization. Streamlines can be defined as one family of grid lines. Lines orthogonal to the the streamlines can be defined as the other family. For example, on a sphere these two families correspond to the two spherical coordinate directions, θ and ϕ, where $x = \cos\theta, y = \sin\phi\cos\theta, z = \sin\phi\sin\theta$. Streamlines have constant θ and the orthogonal lines have constant ϕ.

The sources and sinks provide guidelines on how to distribute the grid lines. The key to distributing grid lines is to highlight the physical features of the domain, that is put more grid lines in regions where interesting physical changes occur. In the flow domain, the most interesting change is the change in direction and velocity of the flow. This change typically occurs most dramatically around the sources and sinks. So, the grid lines should be distributed more densely around them.

The above discussion deals with idealized objects. In the yacht example, there is a keel attached to the hull, and a winglet attached to the keel. The following sections show how to deal with the topological changes in these idealized objects by going through the three gridding steps in more detail.

5 Algorithms

This section presents algorithms that implement the streamline-based gridding ideas discussed in the previous section. The top-level gridding algorithm of Figure 3 is presented again with more detail in Figure 15.

`Partition` takes as input a list of domains, and outputs a list of surface patches. The partitioning lines is computed by the routine `get-partition-lines`, which uses a variety of physical and geometric feature extractors. Then, for each domain it calls `break-surface` to break the domain surface into surface patches. Finally, only the real surface patches are returned and their topological links updated. See Figure 16.

```
1. Partition
   Input: domains = (domain-1, ..., domain-n)
   Output: surface-patches = (patch-BR-1, ..., patch-BR-m)
   partitioning-lines := get-partitioning-lines( domains ).
   For i from 1 to n
       append (break-surface (domain-shape(domain-i),
           partitioning-lines[i])) into all-patches.
   Foreach patch in all-patches
       if (real-surface(patch)) collect patch into surface-patches.
2. parametrize
   Input: surface-patches = (patch-BR-1, ..., patch-BR-m)
   Output: reparam-domains = (param-domain-1, ..., param-domain-m)
   For i from 1 to m
       create-domain(param-domain-i,2).
       mapping-i = create-patch-mapping(patch-BR-i).
       mapping-relation(SHAPE,param-domain-i,patch-BR-i,mapping-i).
       collect (param-domain-i) into repram-domains.
3. distribute
   Input: reparam-domains = (param-domain-1, ..., param-domain-m)
   Output: grid-lines
   For i from 1 to m
       collect ( get-grid-lines(U, param-domain-1),
                 get-grid-lines(V, param-domain-1)) into grid-lines.
```

Figure 15: Gridding algorithm

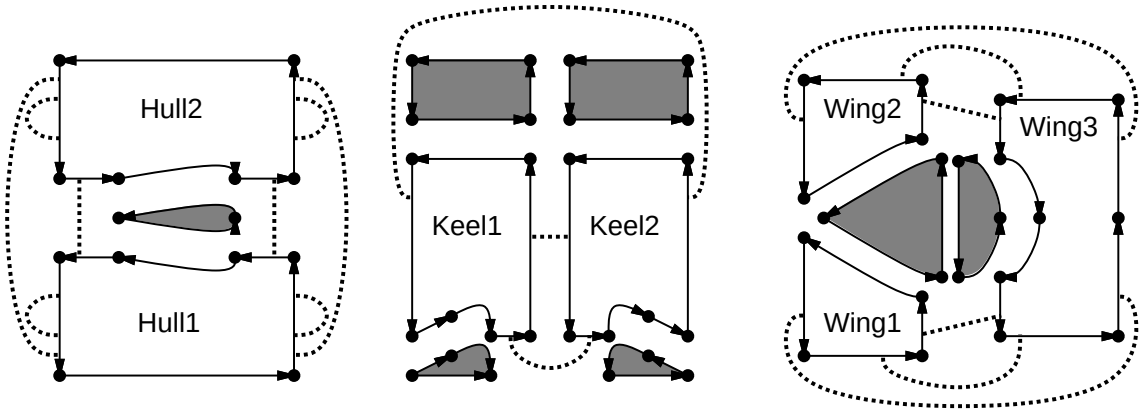

Figure 16: Surface patches after partitioning. Dotted lines across uv-space are not drawn to reduce clutter. Omitted dotted lines would show `Keel1` connected to `Hull1`, `Wing1` and `Wing3`, and would show `Keel2` connected to `Hull2`, `Wing2` and `Wing3`.

`Parametrize` takes as input a list of surface patches, and outputs a list of domains. For each input surface patch `patch-BR-j`, there is a corresponding domain `param-domain-j` that maps to that surface patch. The `create-patch-mapping` command is used to generate this mapping. See Figure 17.

`Distribute` takes as input as list of domains, and outputs the grid lines in parametric coordinates. For each `param-domain-j` domain it calls `get-grid-lines` twice to get $u_{param-domain-j}$ grid lines, then $v_{param-domain-j}$ grid lines. See Figure 17.

This gridding algorithm tries to generate grids that conform to the evaluation criteria. To satisfy the *simple connectedness* constraint `partition` must return simply connected patches. `Get-partitioning-lines` invokes `hole-remover` routine to look for holes. If a hole is found, `hole-remover` invokes `stream-line-tracer` routine to generate stream lines to cut out the hole.

The partition step also has the responsibility of satisfying the *coverage* constraint. To generate grids that only cover the wetted surface areas, `get-partitioning-lines` invokes `surface-surface-intersector` to bound fictional areas of the surface. This enables `partition` to use `real-surface`

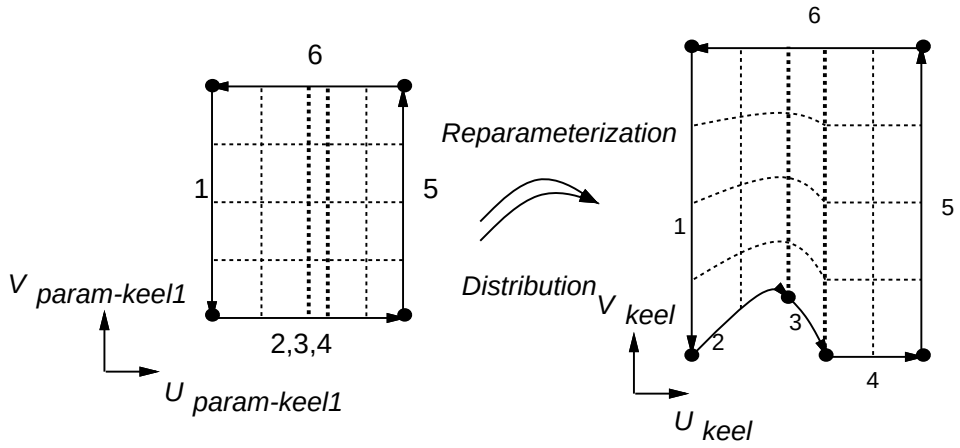

Figure 17: Parametrization and distribution steps for the `Keel1` patch. Parametrization creates `param-keel1` domain and defines mapping from the domain to `Keel1`. The grid lines generated by the distribution step are shown as dotted lines.

routine to remove the fictional surface patches. `Break-surface` returns surface patches that does not overlap nor leave gaps.

`Parameterize` is responsible for the *follow streamlines* and *orthogonality* criteria. The `create-patch-mapping` use a heuristic way of generating the mapping to satisfy the criteria. `Create-patch-mapping` has a much easier job if the the surface patch already follows streamlines and is approximately rectangular, which is the case since `get-partition-lines` tends to generate lines either parallel to the streamlines or orthogonal to the streamlines

`Distribute` is responsible for the *planarity* and *expansion ratio* criteria. *Planarity* implies grid lines should be denser in areas of high curvature. *Expansion ratio* implies that the transition to the denser areas should be smooth. We have experimented with several grid line distribution schemes that tend to satisfy these criteria.

5.1 `Get-partitioning-lines`

The partitioning lines that we use can be divided into three categories: surface-surface intersection lines, streamlines, and source/sink lines.

Surface-surface intersection lines provide the boundary between real and

```
get-partitioning-lines
Input: domains = (domain-1, ..., domain-n)
Output: list-of-list-of-arcs = ( (arcs11, arcs12, ...)
                                 (arcs21, arcs22, ...)
                                 ...
                                 (arcsn1, arcsn2, ...))
```

For each domain, `domain-i`, determine its partitioning lines

1. Find intersection lines by invoking the `surface-surface-intersector`.
2. Invoke `hole-remover` to cut out holes that may have been introduced by the intersection lines. `Hole-remover` calls the `streamline-tracer` to introduce streamlines to make the actual cuts.
3. Invoke `source-sink-line-detector`, which also calls the `streamline-tracer`.

Figure 18: `get-partitioning-lines`

fictional surface areas, so they must be present to satisfy the *coverage* constraint. See Figure 19a.

Notice that the hull-keel intersection line introduces a hole on the hull surface. This hole needs to be cut out, because of the *simple connectedness* constraint. Using streamline-based reasoning, the logical way to "cut" out the hole is by cutting along streamlines, as shown in Figure 19b.

Following streamline-based reasoning source and sink lines are needed. In this case all the sink lines turn out to be redundant. The source lines are shown in Figure 19c. Notice that source/sink nodes in `*XYZ*` may become source/sink lines in *uv* parametric space, as in the hull.

Surface-surface intersection. Surface-surface intersection is a difficult problem and has been extensively studied. Our intersector probably does not push the state of the art, but we include the algorithm here for completeness. Intersection algorithms may be divided into three approaches:

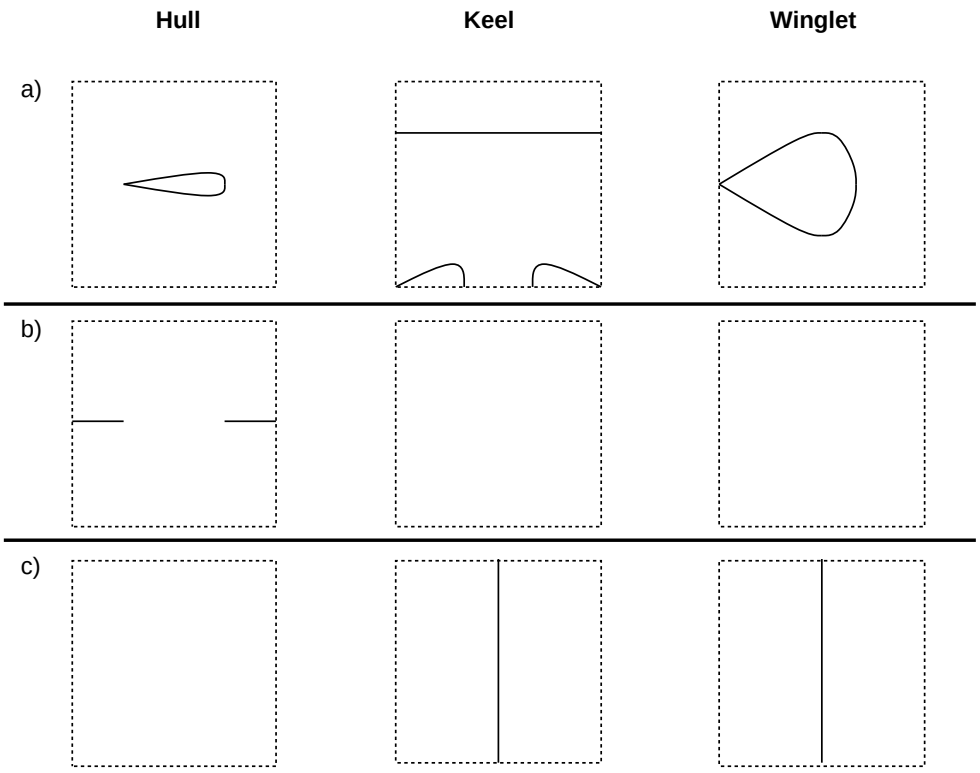

Figure 19: Partition lines: (a) intersection lines, (b) streamlines to "cut" holes out, and (c) source/sink lines.

analytic, divide-and-conquer, and marching. The analytic approach solves for the intersection directly. If an analytic solution of the intersection exists and is efficient to compute, then it is always advantageous. The limitation of this approach is that different algorithms are needed for varying surface types, and sometimes it is inefficient to generate and work with the resulting analytic intersection. For example, the intersection of bicubic polynomial surfaces results in a polynomial of degree 324, see Hoffmann (1989), p.253.

The divide-and-conquer algorithms generally follow these three steps. If the bounding boxes of the surfaces do not intersect, then declare that the surfaces do not intersect. If the two surfaces are approximately planar, then intersect them using a plane-plane intersection routine. Otherwise, divide the surfaces and recursively intersect the subsurfaces. Issues in implement-

ing a divide-and-conquer type algorithm include how to efficiently find true bounding boxes that are not too big, and how to decide if a surface is planar. See Houghton and Emnett (1985), and Natarajan (1990) for examples of this type of algorithm. Generally, divide-and-conquer algorithms are considered to be more robust, but also less efficient, than marching algorithms.

Marching algorithms work by first finding a point on each intersection curve, then for each intersection, marching along it to discover the entire curve. See Barnhill *et al.* (1987) and Müllenheim (1991) for details on such algorithms. Our intersector belongs to the marching class of algorithms. Issues in implementing a marching type algorithm include what space to march in, how to find the first point on the intersection, how to find the direction to march in, how big a marching step to take, and how to represent the intersection curve.

Instead of marching in 3D Cartesian space, we decided to march in $R^4 = (u_1, v_1) \times (u_2, v_2)$ parametric space, generated by combining the parametric spaces of the two surfaces. This way the resulting intersection is actually two curves, one on each of the two surfaces. In BSR terminology these two curves are linked together by a dotted line to show that they are the same curve in 3D Cartesian space.

`Surface-surface-intersector` finds starting points by defining a mapping ($R^4 \rightarrow R$) and minimizing it using steepest descent starting from various initial points in R^4 parametric space. The mapping is defined to be the distance between the Cartesian points of the `SHAPE` mappings. An point on the intersection curve is found when this distance function reaches zero. This method may not detect all the intersections, but probability of finding all the intersections increases with the number of initial points. Currently, we use 16 initial points. Also, to speed up this process the intersector first intersects the bounding boxes of the surfaces to make sure that it is possible for the surfaces to intersect.

The intersector assumes that the intersections of the surfaces are curves, i.e., not points and not sub-surfaces. Once the starting point on a curve is found, the intersector needs to find two directions to march. The above starting point algorithm is called with a point near the starting point to discover a second point on the curve. The second point minus the starting point gives one direction to march. The starting point minus the second point gives the other direction. Once the marching starts, the direction to march is set to be the current point minus the previous point. The next

point on the curve is found by first approximating its location using linear extrapolation, and then by using Newton's method to correct the prediction.

The intersector determines the maximum marching step size as a function of the current curvature of the intersection curve. The current curvature can be determined by the current point and two previous points on the intersection curve. If the maximum step size fails to find the next point, the step size is repeatedly halved until the next point is found or the step size is too small. This variable step size method is efficient, and it is effective in marching around highly curved, but smooth intersections. But, if the curve takes a sharp turn at an acute angle, then the intersector will probably fail to turn fast enough to keep marching along the intersection.

We chose to represent the intersection curves, as well as all the other curve types discussed later, as piecewise-linear curves, instead of some other higher dimensional curves. The principal advantage of a linear representation is simplicity: it is easy to work with. Also, with this representation we know exactly what points on the piecewise-linear curve are on the actual intersection curve that it approximates. If more accuracy is needed, we can always add more line segments to the piecewise-linear curve. Lastly, by the way the curves are generated, high curvature portions of the curves are represented by many short line segments.

Streamlines. The streamline tracer is used in two places: by `hole-remover` to cut out holes in surfaces and by `source-sink-line-detector`, described in the next section, to connect stagnation nodes. The streamline tracer is similar to the marching portion of the surface-surface intersector. Given a current point on a surface, the direction to find the next point is given by the `FLOW` mapping. Also, the step size to take is determined by the curvature of the surface at the current point. The tracing stops when the streamline reaches the boundary of the surface, or when it bumps into a stagnation node.

As mentioned earlier, intersection lines may create holes in surfaces. For each closed intersection line `hole-remover` searches for a leading point and a trailing point along the intersection. From the leading point, we trace a streamline *backward* along the hull surface. From the trailing point, we trace a streamline *forward* along the hull surface. If both of the traced streamlines reach the boundary of the surface the hole is successfully cut out. If one or

both of the lines reach stagnation nodes, then the operation is still successful because of the source/sink lines discussed in the next section.

Since the intersection lines are represented as piecewise linear curves, the leading/trailing points on the actual intersection may not be in the representation. To find the true leading/trailing points `hole-remover` does constrained one-dimensional optimizations to find points that are minimal/maximal with respect to the free stream direction with the constraint that the point is on the actual intersection. Then, these points are added to the intersection curve representation.

Source and sink lines. `Source-sink-line-detector` breaks the source/sink problem into two pieces: for each surface, first it looks for source/sink lines on the boundary of the surface, then it looks for source/sink lines in the interior of the surface.

Half stagnation nodes, i.e., stagnation nodes on the boundary arcs, are relatively easy to locate. For each arc the detector examines the streamline pattern near it. If the streamlines generally flow away from the arc, then it is classified as a source line. If the streamlines generally flow toward the arc, then it is classified as a sink line. Arcs that tend to be parallel to streamlines are either classified as parallel or anti-parallel, depending on the direction the arc is pointing. If the arc is composed of parallel and anti-parallel segments, then the intersection points of these segments must be stagnation nodes.

The detector locates stagnation nodes in the interior of the surface by defining a mapping from points in parametric space to the magnitude of the velocity at that point ($R^2 \rightarrow R$), and minimizing it using steepest descent. The detector obtains starting points for the steepest descent by first examining streamline directions at a coarse array of points. Starting points are picked from areas that show non-uniform flow patterns. The ideal terminating condition is satisfied when the evaluation function becomes zero, that is when a stagnation node has been found. In practice, due to numerical errors, the optimization stops when evaluation function at some non-zero number. In these cases, redundant nodes (several nodes found close together where only one exists) and spurious nodes (nodes found where none should exist) may be introduced. Redundant nodes can be merged into one node. Spurious nodes are dealt with later.

The exact classification of each stagnation node can be determined by

`break-surface`
Input: `surface`
`partition-lines = (arc-1, ..., arc-m)`
Output: `surface-patches`

1. intersect `partitioning-lines` with each other and with the `boundary(surface)` arcs,
2. break all the lines at intersections,
3. form a wire frame from the broken lines,
4. form `surface-patches` based on the wire frame.

Figure 20: `break-surface`

examining a small circle of neighboring flow vectors. For example, if all the neighboring vectors point away from the stagnation node, then it must be a source node. Other stagnation nodes are classified similarly. Based on the object classification, the detector assumes that it is looking for simple lines. For example, Y-shaped line topologies are not allowed. Under this assumption, for any stagnation node there should be a line going through it such that the line is also tangent to a source/sink line at that stagnation node. The magnitude of the flow vector in the *tangent direction* generally tends to be smaller than its neighboring flow vectors in the circle. This is because source/sink lines tend to be orthogonal to the free stream, which implies small fluid movement in the direction tangent of the source/sink lines. However, the off tangent direction is parallel to the free stream, which implies large fluid movement. For half stagnation nodes, instead of a full circle, a half circle is used, and only one tangent direction exists.

The remaining step to find the source/sink lines is to trace streamlines, consisting either of flowing-source nodes or of flowing-sink nodes, to connect the stagnation nodes. Tracing streamlines along flowing-sink nodes is a stable operation. If due to some numerical error the streamline tracer wanders off the flowing-sink path, the flow patterns will force the tracer back on the correct course. The opposite is true for tracing along flowing-source nodes.

The flow patterns near flowing-source nodes exaggerate numerical errors. To solve this problem the detector always runs the streamline tracer backward for flowing-source nodes.

Since the flow vector is zero at a stagnation node, the streamline is traced starting from a point slightly off of a stagnation node in one of its tangent directions, and the streamline must end up close to another stagnation node in one of its tangent directions. This property of streamlines going from stagnation node to stagnation node is very useful in eliminating spurious nodes, since they tend not to link up with other stagnation nodes.

5.2 `Break-surface`

Once the partitioning lines has been constructed, the gridder invokes `break-surface` to get the surface patches, see Figure 20.

Step one is straightforward to accomplish. We use a divide-and-conquer type algorithm to intersect these lines, because finding bounding boxes for a piecewise linear curve is easy and determining if it is linear is trivial.

In step two, curves sewn to the partitioning lines need to be broken, so that this `sewn` relationship may be propagated to the broken lines. For example, breaking the hull intersection line in Figure 19a implies breaking the corresponding keel intersection line.

In step three, the broken partitioning lines are converted to pairs of arcs that point in opposite directions. These arc pairs are sewn together since they map to the same edge in 3D Cartesian space. Then, shortest possible, simple, and counter-clockwise loops are extracted from the arcs. Here by "simple" we mean that each arc is used once and only once. Each counter-clockwise loop implicitly binds the surface area within it.

In step four, these loops are converted to surface patches using `make-implicit`. The union of the surface patches is the original unit square, and the surface patches only intersect each other at the boundaries.

The surface patches after partitioning are shown in Figure 16. The shaded surface patches are fictional and will not be gridded.

5.3 `real-surface`

Real and fictional surface patches can be distinguished by reasoning using the outward `NORMAL` mapping, the *counter-clockwise* rule, and intersection lines.

```
real-surface
Input: surface-patch
Output: True or False
Return True if
for each arc1 in boundary(surface-patch)
   arc2 := sewn-to(arc1)
   neighbor-patch := boundary-of(arc2)
   the inside direction of arc1 points in the same direction as
        get-mapping (NORMAL,domain(neighbor-patch),*xyz*)
```

Figure 21: `real-surface`

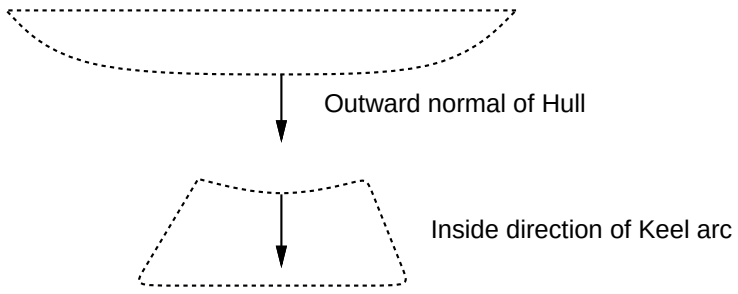

Figure 22: Reasoning about real and fictional surface patches. The outward normal of the hull points in the same direction as the inside direction of the keel arc implies the keel is outside the hull.

`create-patch-mapping`
Input: `surface-patch`
Output: `mapping`

1. Divide the boundary arcs, `boundary(surface-patch)`, into four groups: two orthogonal to the streamlines, one parallel, and one anti-parallel to the the streamline.

2. Define mapping using transfinite interpolation by interpolating from one orthogonal group to the other orthogonal group, and by interpolating from the parallel group to the anti-parallel group in the other direction.

Figure 23: `create-patch-mapping`

For example, the surface patch `Keel1` has a intersection line in common with the surface patch `Hull1`. Along this intersection the outward normals of `Hull1`, `get-mapping (NORMAL,domain(Hull1),*xyz*)`, generally point in the negative z-direction. Using the *counter-clockwise* the inside direction of arc on `Keel1` points in the negative V_{keel} direction, which also corresponds to the negative z-direction in `*XYZ*`, see Figure 22. This implies that `Keel1` is outside of `Hull1`. Similar reasoning shows that `Keel1` is outside of `Wing1` and `Wing3`. Since `Keel1` is on the outside of all its neighboring surfaces, `Keel1` is a real surface patch. If a surface patch is on the inside of one or more of its neighbors, then it is a fictional patch.

5.4 `create-patch-mapping`

`Create-patch-mapping` uses a mapping technique called *transfinite interpolation*, see Thompson *et al.* (1985), and Knupp and Steinberg (1993). Given a patch bounded by four curves, transfinite interpolation maps a unit square onto the patch by interpolating from opposite curves of that quadrilateral. This method requires the surface patch to be parametrized to have exactly *four sides*. But surface patches tend to have more than four boundary arcs. In order to use transfinite interpolation, `Create-patch-mapping` uses a heuris-

tic, streamlined-based method to group the boundary arcs. See Figure 17.

The orientation with respect to the streamlines of each arc is classified as either parallel, anti-parallel, or orthogonal. For example, the patch `Keel1` is bounded by six arcs. Arc 1 is a sink line. Arc 5 is a source line. So, by definition they are orthogonal to the streamlines. Arc 4 is a boundary arc from the original input surface. Arcs 2, 3 and 6 are intersection lines. These four arcs are neither completely parallel nor completely orthogonal to the streamlines. But by sampling different segments of these arcs, we can approximately classify arcs 2 and 4 as parallel, 6 as anti-parallel, and arc 3 as orthogonal. So, six groups are formed, $\{(1),(2),(3),(4),(5),(6)\}$. But, unlike the graphical depiction in Figure 17, arc 3 is very short when compared to its neighbors, arc 2 and 4. So, heuristically merging arc 3 with its neighbors, we get four groups, $\{(1),(2,3,4),(5),(6)\}$.

Our grouping method works well, because the boundary arcs of the surface patches tend to be partitioning lines: intersection lines, streamlines, and source/sink lines. Classification of streamlines and source/sink lines are straightforward. In practice, intersection lines tend always to be parallel, because an orthogonal intersection line causes too much drag, and would not be used in properly designed streamlined body.

This heuristic method may fail to group the boundary arcs into four groups. Failure indicates that the geometry of the surface patch is too complicated, and additional partitioning lines may be needed. So far we have not encountered such a case.

5.5 `get-grid-lines`

According to streamline-based reasoning, grid lines should be concentrated more densely around sources and sinks. In our streamline-based gridding method sources and sinks tend to be at the ends of the surface patches (in Figure 17, arc 1 and arc 5). So, complicated distribution schemes usually are not needed. We have experimented with *equal-arc*, *cosine* and *hyperbolic tangent* schemes, which distribute more grid lines at the ends and yet distribute them smoothly enough to avoid violating the *expansion ratio* constraint. All three schemes work well, but if many grid lines are laid out, *cosine* tends to place grid lines too densely at the ends. This leads to numerical round-off error.

Beside resolving physical features, distribution must also resolve geomet-

`get-grid-lines`
Input: `coordinate, domain`
Output: {`coordinate = (constants)`}

1. Lay down the required grid lines
   (a) Lay down grid lines at the boundaries: `coordinate = 0` and `coordinate = 1`
   (b) Lay down grid lines at arc-arc intersections
2. Lay down enough grid lines in between the required grid lines to satisfy the `grid-line-density` parameter, which is specified by the user.

Figure 24: `get-grid-lines`

ric features. For example, in Figure 17 one $u_{param_{keel1}} = constant$ grid line must be laid out at the intersection of arc 2 and arc 3, and another one at the intersection of arc 3 and arc 4. Grid lines that must be laid out are shown as heavy dotted lines. The node at the intersection of arc 3 and arc 4 touches three surface patches, `Keel1`, `Keel2`, and `Wing3`. Not laying a grid line at that node would create a gap there so the three patches would not meet.

6 Computational Results

The purpose of these computational experiments is to test if the program is able to generate good grids for a variety of geometries. Here we present the results of three such experiments: yacht with hull, keel and winglet; yacht with hull and two keels; and yacht with hull, keel, and three winglets. For more experiments see Yao (1996).

Our gridding program is able to automatically generate grids for each of the three yacht configurations. See Figures Figure 2 and 25. Using simplified projection flow model of the physics and using knowledge of geometry, our program is able to recognize and construct relevant gridding features: source/sink lines, streamlines, and surface intersection lines. Then, using

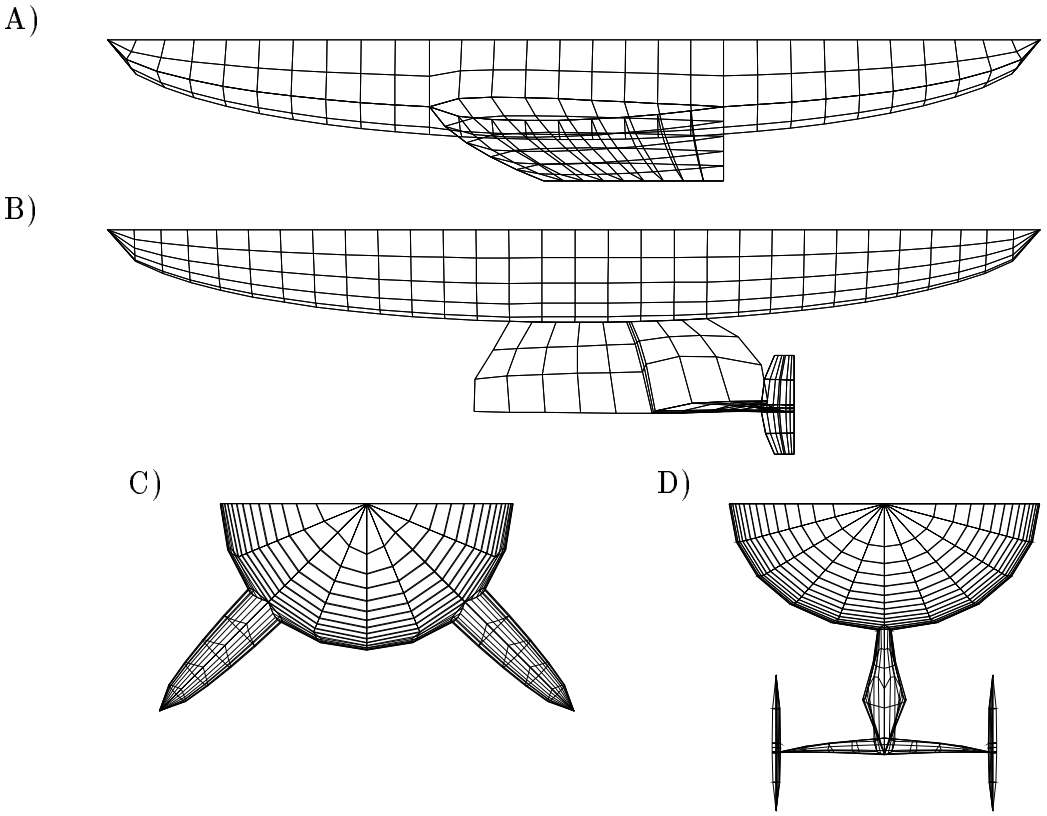

Figure 25: A) Side view of yacht with hull and two keels. B) Side view of yacht with hull, keel and three winglets. C) Front view of A. D) Front view of B.

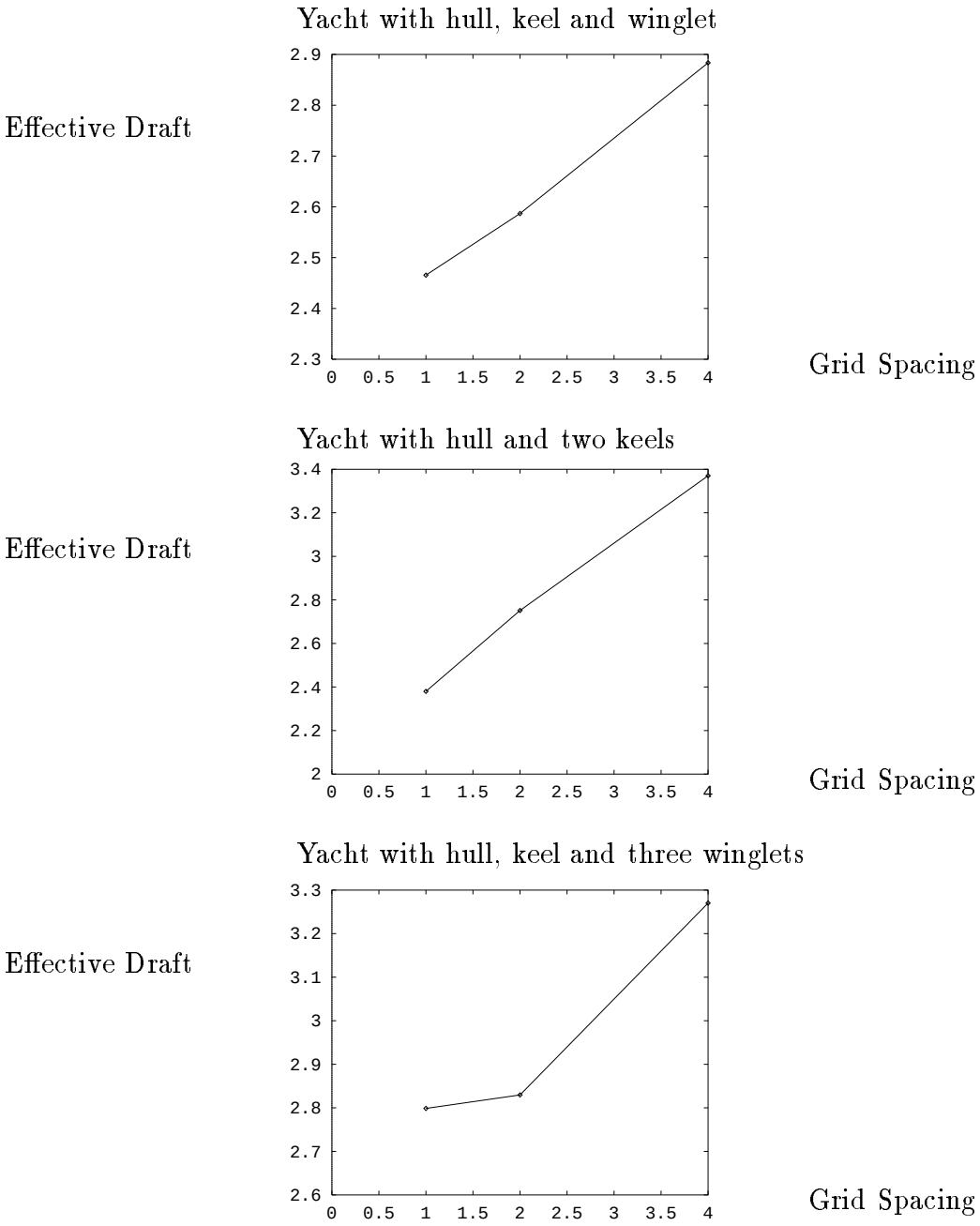

Figure 26: Convergence study of Effective draft

these feature lines the grid partitioning is formed by constructing the surface patches and neighbor relation. This partitioning is formed without the need of human intervention or the need of pre-processing the input. The parametrization and distribution are also done automatically.

For each yacht configuration a convergence study is performed to check the goodness of the grid. A convergence study is a series of PMARC simulations using grids with the same partitioning and parametrization, but the grid spacing of the each successive grid in the series is halved. As the density of a good grid increases, output quantities computed by PMARC should converge to their correct values. In designing yachts the most important output quantity is *effective draft*, a measure of the efficiency of a sailing yacht's keel. See Figure 26 for plots of effective drafts versus grid spacings. The plots show that the effective drafts are converging, that is the differences in successive effective drafts are decreasing as the grid spacings are decreased.

Other values can also be used to check the soundness of the simulation. One such value is the pressure coefficient (C_p), which measures flow vector velocity at surface points. A pressure coefficient of one implies the velocity is zero. As the pressure coefficient decreases the velocity increases. A pressure coefficient of zero implies the velocity is the same as the free stream velocity. To resolve stagnation nodes the maximum C_p over the entire surface should approach one as the grid spacing is decreased. To rule out non-physical flow velocities the minimum C_p over the entire surface should not be too negative. Large negative values usually indicate flaws in the grid. For example, for the yacht with hull, keel and winglet example Figure 27 shows the maximum C_p approaching one, and the minimum C_p not growing too negative. See Gelsey (1995) for discussions on automated evaluation of simulation output quality.

Grid Spacing	Panels	Lift	Drag	Draft	min C_p	max C_p
4	286	-.288	.095	2.884	-3.183	0.619
2	938	-.253	.125	2.587	-4.673	0.794
1	3678	-.242	.148	2.466	-4.128	0.901

Figure 27: Convergence study of yacht with hull, keel and winglet.

7 Future Work

This work can be extended in various directions. One direction is to add feedback and local refinement capabilities to the gridder. The streamlines predicted by PMARC may be fed back into the gridder to improve the grid. Also, the gridder can be extended to detect and correct local flaws in the grid based on intermediate values, such as the coefficient of pressure. Another direction is to extend the gridder to other physical domains where PDE simulators are needed. We believe our methodology of identifying key physical features of the domain and of reasoning about how they interact with the geometry is quite general and extensible. For example, in the ingot casting problem of heat transfer the temperature profile seems to be the key feature, see Ling *et al.* (1993). Temperature profiles tend to change the fastest near sharp corners and in appendages (regions where the surface area to volume ratio is large). This suggests that isotherms should be useful as grid lines, and they should be distributed more densely near corners and appendages.

8 Related Work

Using streamlines is a natural idea. Chung *et al.* (1993) define a 2D finite-difference method based on streamline-coordinates, instead of Cartesian coordinates. Chao and Liu (1991) apply streamline-based gridding to 2D flow problems consisting of a single patch. Many geometric modeling systems have been developed, such as Alpha1 in Riesenfeld (1981) and SHAPES in Sinha (1992). Requicha (1980) provides a good survey of the theoretical foundations of solid modeling. Most of these systems are intended for modeling mechanical components, and provide little support for gridding as they lack capabilities like representation of parametric space objects for parametrization and distribution, and algorithms to manipulate these objects.

Previous AI work in gridding includes Santhanam *et al.* (1992), Dannenhoffer (1992), and Vogel (1990). Santhanam identifies several key parameters to modify and improve grids for 1D Euler simulations. In the 2D planar flow domain both Dannenhoffer and Vogel created systems that are capable of doing partitioning. Beside the different application domain (planar vs surface grids), their approaches to gridding differ philosophically from our approach. Dannenhoffer argues that gridding around 2D bodies is difficult, so an au-

tomated gridder needs to modify previous gridded bodies to fit the current situation. Thus, Dannenhoffer is taking a case-based approach. Vogel also believes that no theory of partitioning exists, so she models her gridder after experts using a knowledge-based approach. Each primitive body is instantiated from one of four qualitative shapes. Then, using expert system style rules the primitive bodies are grouped into subgroups. Finally, based upon the shape classification, the grouping, and more expert rules, the gridder heuristically designs a partitioning. Our approach may be classified as a model-based reasoning approach. We reason using knowledge of physics, knowledge of numerical analysis, and knowledge of how they interact with geometry to generate the grid.

Streamlines in non-potential flow domains involving vorticities and separations can be quite complicated. Visualization work by Helman and Hesselink (1990), and Globus *et al.* (1991) involve extracting streamline topologies from numerical simulation data to better understand and interpret the streamlines. More detailed analysis of stagnation points and flow patterns are found in Tobak and Peake (1982), and Perry and Chong (1987). In potential flow the streamline topologies are simple enough to predicate *a priori* (at least simple enough to the accuracy needed for gridding), without the need for topology extraction from numerical simulation. An alternative is to first generate a simple, default grid, then run the simulator, then extract the topology from the simulation.

The topology extraction part of our gridder is reminiscent of the phase space analysis work by Sacks (1991), Yip (1991), and Zhao (1991). Perhaps similarity is not surprising since much of the topology work in fluid dynamics has been inspired by the success of Poincaré's work in dynamic systems.

For a shorter conference paper version of this article see Yao and Gelsey (1994).

9 Conclusion

Numerical simulation of partial differential equations is a powerful tool for engineering design. However, PDE solvers require as input a grid, which requires considerable time and effort to generate. We have developed an intelligent automated system that is able to generate the grid directly from a geometric description of the physical situation. No human pre-processing of the input into a partitioning of surface patches is needed. This enables our

gridding system to handle a wide range of geometric configurations.

The system works by exploiting and operationalizing specific knowledge related to gridding. The system incorporates three types of knowledge: knowledge of numerical analysis, knowledge of physics, and knowledge of geometry.

- *Knowledge of numerical analysis* includes both specific knowledge of the PDE solver (PMARC) and general numerical knowledge.
  - Surface patches must be *simply connected* if the grid is represented as arrays of corner points.
  - A grid should exactly *cover* the spatial domain of interest.
  - *Planar* panels more faithfully represent the spatial domain.
  - *Orthogonal* grids reduce numerical differentiation error.
  - Large *expansion ratio* increases numerical differentiation error.
  - Grids capturing the physics of the flow have better numerical convergence properties.
- *Knowledge of physics* includes a simplified model of the domain, and how it interacts with the geometry.
  - Simplified model of potential flow: projection flow model
  - Point features (surface point classification): stagnation nodes (source, sink, saddle, source-saddle, sink-saddle), and non-stagnation nodes (uniflow, flowing-source, flowing-sink)
  - Aggregate line features: streamline, source line, sink line
  - The streamline topology constructed from these features captures the essence of the flow.
- *Knowledge of geometry* includes knowledge of to represent geometric entities, how to manipulate the representation, and how to detect features.
  - Geometry of surfaces represented by mappings from parametric space to Cartesian space.

- Topology represented by surface patches and neighbor relations
- Break surface into surface patches
- Remove fictional surface areas
- Cut out holes in surfaces surfaces
- Physical feature detectors: point features and aggregate line features
- Geometrical feature detector: surface-surface intersection line

This article contributes to model-based reasoning, spatial reasoning, and intelligent scientific computation in the following ways:

- This article demonstrates how simple models may be used to predict the approximate behaviors of a physical system and how qualitative (topological) information may be deduced from the approximate behavior to aid in obtaining more accurate behaviors from complex (PDE) models.
- Imagistic reasoning and visualization play important roles in gridding and more generally in problem solving and in reasoning about physical systems. In the gridding domain this article presents a program that exhibits these abilities by making use of physical domain knowledge and geometric knowledge.
- The gridder in this article may be classified as a second generation AI gridding system as defined by Andrews (1987). It is capable of automated construction and synthesis of a grid, instead of classifying and selecting from pre-enumerated solutions.

10 Acknowledgments

The research was done in consultation with Rutgers Computer Science Dept. faculty member Gerard Richter. We worked with hydrodynamicists Martin Fritts and Nils Salvesen of Science Applications International Corp., and John Letcher of Aero-Hydro Inc. Our research benefited significantly from interaction with the members of the Rutgers AI/Design group. Thanks to Nathalie Japkowicz for proofreading the paper. This research was partially supported by NSF grant CCR-9209793, ARPA/NASA grant NAG2-645, and ARPA contract ARPA-DABT 63-93-C-0064.

References

[Andrews 1987] Alison E. Andrews. Progress and challenges in the application of artificial intelligence to computational fluid dynamics. *AIAA Journal*, 26:40–46, 1987.

[Barnhill *et al.* 1987] R. E. Barnhill, G. Farin, M Jordan, and B. R. Piper. Surface/surface intersection. *Computer Aided Geometric Design*, 4:3–16, 1987.

[Chao and Liu 1991] Y. C. Chao and S. S. Liu. Streamline adaptive grid method for complex flow computation. *Numerical Heat Transfer, Part B*, 20:145–168, 1991.

[Chung *et al.* 1993] S. G. Chung, K. Kuwahara, and O. Richmond. Streamline-coordinate finite-difference method for hot metal deformations. *Journal of Computational Physics*, 108:1–7, 1993.

[Dannenhoffer 1992] John F. Dannenhoffer. Automatic block-structured grid generation — progress and challenge. In *Intelligent Scientific Computation*, number FS-92-01 in AAAI Technical Reports. AAAI Press, 1992.

[Ellman *et al.* 1992] Tom Ellman, John Keane, and Marc Schwabacher. The Rutgers CAP Project Design Associate. Technical Report CAP-TR-7, Department of Computer Science, Rutgers University, 1992.

[Gelsey 1995] Andrew Gelsey. Intelligent automated quality control for computational simulation. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing (AI EDAM)*, 9(5):387–400, November 1995.

[Globus *et al.* 1991] A. Globus, C. Levit, and Lasinski. A tool for visualizing the topology of three-dimensional vector fields. In *IEEE Conference on Visualization*, pages 33–40. IEEE Computer Society Press, 1991.

[Helman and Hesselink 1990] J. L. Helman and Lambertus Hesselink. Surface representations of two- and three-dimensional fluid flow topology. In *IEEE Conference on Visualization*, pages 6–13. IEEE Computer Society Press, 1990.

[Hess 1990] J. L. Hess. Panel methods in computational fluid dynamics. *Annual Review of Fluid Mechanics*, 22:255–274, 1990.

[Hoffmann 1989] Christoph M. Hoffmann. *Geometric and solid modeling.* Morgan Kaufmann, 1989.

[Houghton and Emnett 1985] Elizabeth G. Houghton and Robert F. Emnett. Implementation of a divide-and-conquer method for intersection of parametric surfaces. *Computer Aided Geometric Design*, 2:173–183, 1985.

[Kao and Su 1992] T. J. Kao and T. Y. Su. An interactive multi-block grid generation system. In Robert E. Smith, editor, *Software Systems for Surface Modeling and Grid Generation*, number 3143 in NASA Conference Publication, pages 333–345, April 1992.

[Katz and Plotkin 1991] Joseph Katz and Allen Plotkin. *Low-speed aerodynamics: from wing theory to panel methods.* McGraw-Hill, 1991.

[Knupp and Steinberg 1993] Patrick M. Knupp and Stanly Steinberg. *The fundamentals of grid generation.* CRC Press, 1993.

[Ling *et al.* 1993] Sui-ky Ringo Ling, Louis Steinberg, and Yogesh Jaluria. MSG: A computer system for auotmated modeling of heat transfer. *Artificial Intelligence for Enginnering Design, Analysis and Manufacturing (AI EDAM)*, 7(4):287–300, 1993.

[Müllenheim 1991] Gregor Müllenheim. On determining start points for a surface/surface intersection algorithm. *Computer Aided Geometric Design*, 8:401–408, 1991.

[Narayanan 1993] N. Hari Narayanan. Imagery: Computational and cognitive perspectives. *Computational Intelligence*, 9(4):303–307, 1993.

[Natarajan 1990] B. K. Natarajan. On computing the intersection of b-splines. In *Proceedings of the 6th Annual ACM Symposium on Computational Geometry*, pages 157–167, 1990.

[Perry and Chong 1987] A. E. Perry and M. S. Chong. A description of eddying motions and flow patterns using critical-point concepts. *Annual Review of Fluid Mechanics*, 19:125–155, 1987.

[Remotique *et al.* 1992] M. G. Remotique, E. T. Hart, and M. L. Stokes. EAGLEView: A surface and grid generation program and its data management. In Robert E. Smith, editor, *Software Systems for Surface Modeling and Grid Generation*, number 3143 in NASA Conference Publication, pages 243–251, April 1992.

[Requicha 1980] Aristides A. G. Requicha. Representations for rigid solids: Theory, methods, and systems. *Computing Surveys*, 12(4):437–464, Dec 1980.

[Riesenfeld 1981] R. F. Riesenfeld. Using the oslo algorithm as a basis for CAD/CAM geometric modeling. In *Proc. NCGA National Conf.*, pages 345–356, Fairfax, Va, 1981.

[Sacks 1991] Elisha P. Sacks. Automatic analysis of one-parameter planar ordinary differential equations by intelligent numeric simulation. *Artificial Intelligence*, 48:27–56, 1991.

[Santhanam *et al.* 1992] Tharini Santhanam, J.C. Browne, J. Kallinderis, and D. Miranker. A knowledge based approach to mesh optimization in CFD domain: 1D Euler code example. In *Intelligent Scientific Computation*, number FS-91-01 in AAAI Technical Reports. AAAI Press, 1992.

[Schuster 1992] David M. Schuster. Batch mode grid generation: An endangered species? In Robert E. Smith, editor, *Software Systems for Surface Modeling and Grid Generation*, number 3143 in NASA Conference Publication, pages 487–500, April 1992.

[Sinha 1992] Pradeep Sinha. Mixed dimensional objects in geometric modeling. In *New Technologies in CAD/CAM*, 1992.

[Thompson *et al.* 1985] Joe F. Thompson, Z. U. A. Warsi, and C. Wayne Mastin. *Numerical Grid Generation : Foundations and Applications.* North-Holland, Amsterdam, 1985.

[Tobak and Peake 1982] Murray Tobak and David J. Peake. Topology of three-dimensional separated flows. *Annual Review of Fluid Mechanics*, 14:61–85, 1982.

[Vogel 1990] Alison Andrews Vogel. Automated domain decomposition for computational fluid dynamics. *Computers and Fluids*, 18(4):329–346, 1990.

[Yao and Gelsey 1994] Ke-Thia Yao and Andrew Gelsey. Intelligent automated grid generation for numerical simulations. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 1224–1230, 1994.

[Yao 1996] Ke-Thia Yao. *Intelligent Automated Grid Generation for Numerical simulations*. PhD thesis, Rutgers University, 1996. To be published.

[Yip 1991] Kenneth Man-Kam Yip. Understanding complex dynamics by visual and symbolic reasoning. *Artificial Intelligence*, 51:179–221, 1991.

[Zhao 1991] Feng Zhao. Extracting and representing qualitative behaviors of complex systems in phase spaces. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, 1991.