SANDIA REPORT

SAND2007-0525
Unlimited Release
Printed April 2007

A Data Storage Model for Novel Partial Differential Equation Discretizations

David Thompson, Philippe P. Pébay, and Wendy S. K. Doyle

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94-AL85000.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.doe.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/ordering.htm

SAND2007-0525
Unlimited Release
Printed April 2007

A Data Storage Model for Novel Partial Differential Equation Discretizations

David Thompson
Sandia National Laboratories
M.S. 9152, P.O. Box 969
Livermore, CA 94550, U.S.A.
dcthomp@sandia.gov

Philippe P. Pébay
Sandia National Laboratories
M.S. 9051, P.O. Box 969
Livermore, CA 94550, U.S.A.
pppebay@sandia.gov

Wendy S. K. Doyle
Sandia National Laboratories
M.S. 9152, P.O. Box 969
Livermore, CA 94550, U.S.A.
wkoegle@sandia.gov

Abstract

The purpose of this report is to define a standard interface for storing and retrieving novel, non-traditional partial differential equation (PDE) discretizations. Although it focuses specifically on finite elements where state is associated with edges and faces of volumetric elements rather than nodes and the elements themselves (as implemented in ALEGRA), the proposed interface should be general enough to accomodate most discretizations, including *hp*-adaptive finite elements and even mimetic techniques that define fields over arbitrary polyhedra. This report reviews the representation of edge and face elements as implemented by ALEGRA. It then specifies a convention for storing these elements in EXODUS files by extending the EXODUS API to include edge and face blocks in addition to element blocks. Finally, it presents several techniques for rendering edge and face elements using VTK and ParaView, including the use of VTK's generic dataset interface for interpolating values interior to edges and faces.

Acknowledgement

The authors would like to thank the LDRD Senior Council for the opportunity to pursue this research. The authors were supported by the United States Department of Energy, Office of Defense Programs by the Laboratory Directed Research and Development Senior Council, project 90499. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed-Martin Company, for the United States Department of Energy under contract DE-AC04-94-AL85000.

Contents

Figures

This page left intentionally blank

A Data Storage Model for Novel Partial Differential Equation Discretizations

1 Introduction

Partial differential equation (PDE) solution techniques (finite elements, finite differences, finite volumes, meshless methods, boundary element methods, moving mesh methods, and others) have been developing continuously and rapidly for some time. The states that these techniques store are unique, but they share a common property: each performs some discretization of the original domain of the PDE and this discretization provides finite sets that can be enumerated on a computer. These sets associated with the discretizations are stored in memory and on disk using structures basic to computer science: linear arrays, multidimensional arrays, arrays of arrays (whose interior entries vary in size unlike multidimensional arrays), and so forth. This document defines a disk storage model for novel PDE discretizations. One specific discretization targeted by this storage model is the De Rham complex shape functions employed by ALEGRA, and that will be used as a running example throughout the document.

The ALEGRA computing framework provides electromagnetic simulation on arbitrary quadrilateral and hexahedral grids with exact sequences of finite element spaces that mimic the mathematical structure of Maxwell's equations. These elements have, in particular, edge circulations and surface fluxes, that cannot currently be handled by existing visualization applications. In this report, we briefly review the main features of these elements, and then discuss how the Sandia Higher Order Elements framework can be used to visualize them in ParaView.

1.1 Hexahedral Elements

Throughout this report, we use the following notation: $\widehat{K} = [-1,1]^3$ is the domain of the reference hexahedron's parametric coordinates, K is the domain of the physical coordinates of a particular hexahedral element, $F : \widehat{K} \mapsto K$ is the homeomorphism that maps from reference to physical coordinates, G denotes F^{-1}. α, β and γ are arbitrary numbers in $\{-1,1\}$, the triplet (i,j,k) is an even permutation of $(1,2,3)$. With this convention, we define:

- $\xi_{ijk}^{\alpha\beta\gamma}$ to be the point (corner node) in $\widehat{K}$ with parametric coordinates $\xi_i = \alpha$, $\xi_j = \beta$, and $\xi_k = \gamma$. There is no risk of ambiguity so the $_{ijk}$ subscript can be made implicit and the notation $\xi^{\alpha\beta\gamma}$ used instead;

- $\xi_{ij}^{\alpha\beta}$, the segment (edge) in $\widehat{K}$ parameterized by $\xi_i = \alpha$, $\xi_j = \beta$, and $\xi_k \in [-1,1]$;
- ξ_i^{α}, the planar quadrangle (face) in $\widehat{K}$ parameterized by $\xi_i = \alpha$ and $(\xi_j, \xi_k) \in [-1,1]^2$.

Let $x^{\alpha\beta\gamma} = F(\xi^{\alpha\beta\gamma})$, $x_{ij}^{\alpha\beta} = F(\xi_{ij}^{\alpha\beta})$, and $x_i^{\alpha} = F(\xi_i^{\alpha})$ – these are, respectively, the vertices, edges, and faces of K. Using the linear Lagrange interpolants for each parametric coordinate, but written in terms of physical coordinates, *i.e.*:

$$\phi_i^{\alpha} = \frac{1+\alpha G_i}{2}$$

for each choice of i in $\{1,2,3\}$, we define the following sets of functions over K:

- the standard trilinear shape functions, of which there are 8: $W_{ijk}^{\alpha\beta\gamma} = \phi_i^{\alpha}\phi_j^{\beta}\phi_k^{\gamma}$;
- the edge circulation shape functions, of which there are 12: $W_{ij}^{\alpha\beta} = \phi_i^{\alpha}\phi_j^{\beta}\nabla\phi_k^{\gamma}$;
- the face flux shape functions, of which there are 6: $W_i^{\alpha} = \phi_i^{\alpha}\nabla\phi_j^{\beta} \times \nabla\phi_k^{\gamma}$;
- the volume scalar shape function, of which there is 1: $W = \nabla\phi_i^{\alpha} \cdot \nabla\phi_j^{\beta} \times \nabla\phi_k^{\gamma} = \det(\nabla\phi_i^{\alpha}, \nabla\phi_j^{\beta}, \nabla\phi_k^{\gamma})$,

These four sets of functions respectively span the function spaces denoted by $\mathscr{W}^0(K)$, $\mathscr{W}^1(K)$, $\mathscr{W}^2(K)$, and $\mathscr{W}^3(K)$. It can be shown (*cf.* [1]) that the $\mathscr{W}^i(K)$s form the following exact sequence

$$\mathscr{W}^0(K) \overset{\nabla}{\longmapsto} \mathscr{W}^1(K) \overset{\nabla\times}{\longmapsto} \mathscr{W}^2(K) \overset{\nabla\cdot}{\longmapsto} \mathscr{W}^3(K)$$

that provides suitable approximation of the De Rham complex associated with Maxwell's equations.

Bocev et al. [1] note some interesting properties of these functions:

- since $\nabla\phi_j^{\gamma} = \gamma\nabla G_j$, then $W_{ij}^{\alpha\beta} = \gamma\phi_i^{\alpha}\phi_j^{\beta}\nabla\|\nabla G_k^{\alpha}\|n_k^{\gamma}$, and thus $W_{ij}^{\alpha\beta}$ is normal to the two faces x_k^{γ} at its endpoints;
- since $\nabla\phi_j^{\beta} \times \nabla\phi_k^{\gamma}$ is a vector tangent to edge $x_{jk}^{\beta\gamma}$, then so is W_i^{α};
- since $[\nabla\phi_i^{\alpha}, \nabla\phi_j^{\beta}, \nabla\phi_k^{\gamma}]$ is the volume of the parallelepiped with edge vectors $\nabla\phi_i^{\alpha}$, $\nabla\phi_j^{\beta}$ and $\nabla\phi_k^{\gamma}$], then W is a scalar proportional to the angle between edge $x_{jk}^{\beta}\gamma$ (resp. $x_{ik}^{\alpha}\gamma$, $x_{ij}^{\alpha}\beta$) and the normal to x_i^{α} (resp. x_j^{β}, x_k^{γ}).

The 12 vector basis functions $W_{ij}^{\alpha\beta}$ are shown in Figure 1 for an arbitrary hexahedron. This figure makes several properties clear. For example if t is the vector tangent to $x_{ij}^{\alpha\beta}$, then $\int_{x_{ij}^{\kappa\mu}} W_{ij}^{\alpha\beta}(x) \cdot t d\ell = 0$ for all edges except $x_{ij}^{\kappa\mu} = x_{ij}^{\alpha\beta}$ clearly holds, but that does not require $W_{ij}^{\alpha\beta}$ to be tangent to edge $x_{ij}^{\alpha\beta}$.

The application (magnetic diffusion) being considered uses standard isoparametric hexahedra (*i.e.*, trilinear) and thus the geometric map from reference to physical coordinates within element K is simply

$$F_K = \sum_{(\alpha,\beta\gamma)\in\{-1,1\}^3} x^{\alpha\beta\gamma} \widehat{W}_{ijk}^{\alpha\beta\gamma},$$

i.e., the Lagrange tensor product interpolation scheme.

1.2 Quadrilateral Elements

An exact sequence of finite elements spaces on quadrilateral elements can be constructed by specializing the discussions above: a quadrilateral K is embedded into a virtual hexahedral $\tilde{K} = K \times [-1,1]$ (*cf.* [1] for details). In this case, $\widehat{K}$ denotes $[-1,1]^2$, and F is a map from $\mathbb{R}^2$ to itself, and the hexahedral results become, after specialization:

- $W_{ij*}^{\alpha\beta*} = \phi_i^\alpha \phi_j^\beta$;
- $W_{ij}^{\alpha*} = \phi_i^\alpha \nabla\phi_j^\beta$;
- $W_i^\alpha = \phi_i^\alpha \nabla\phi_j^\beta \times \mathbf{k}$;
- $W = \det(\nabla\phi_i^\alpha, \nabla\phi_j^\beta, \frac{\mathbf{k}}{2})$,

where $\mathbf{k}$ denotes the third basis vector in the orthonormal physical basis.

These interpolants can be illustrated by considering the quadrilateral element Ω^a of Figure 2. To the right are four figures, each with dashes showing the direction $\vec{w}_e$ for the associated edge e. The corresponding edge e has a vector drawn with a thick black line, its sign and magnitude defined by Γ_e. Note that the basis for each edge is tangential to that edge and normal to all the other edges in the element. This means that tangential continuity across elements is guaranteed while normal continuity is not.

In order to reconstruct the field $\vec{w}$ for some element domain Ω^a, we need the four Γ_e values on its borders. Since neighboring elements (say Ω^b) may share some edges with Ω^a, and since shared edges may be reversed with respect to their neighbors, we must be careful to define the Γ_e so that the interpolations of $\vec{w}$ to Ω^a and Ω^b are tangentially continuous

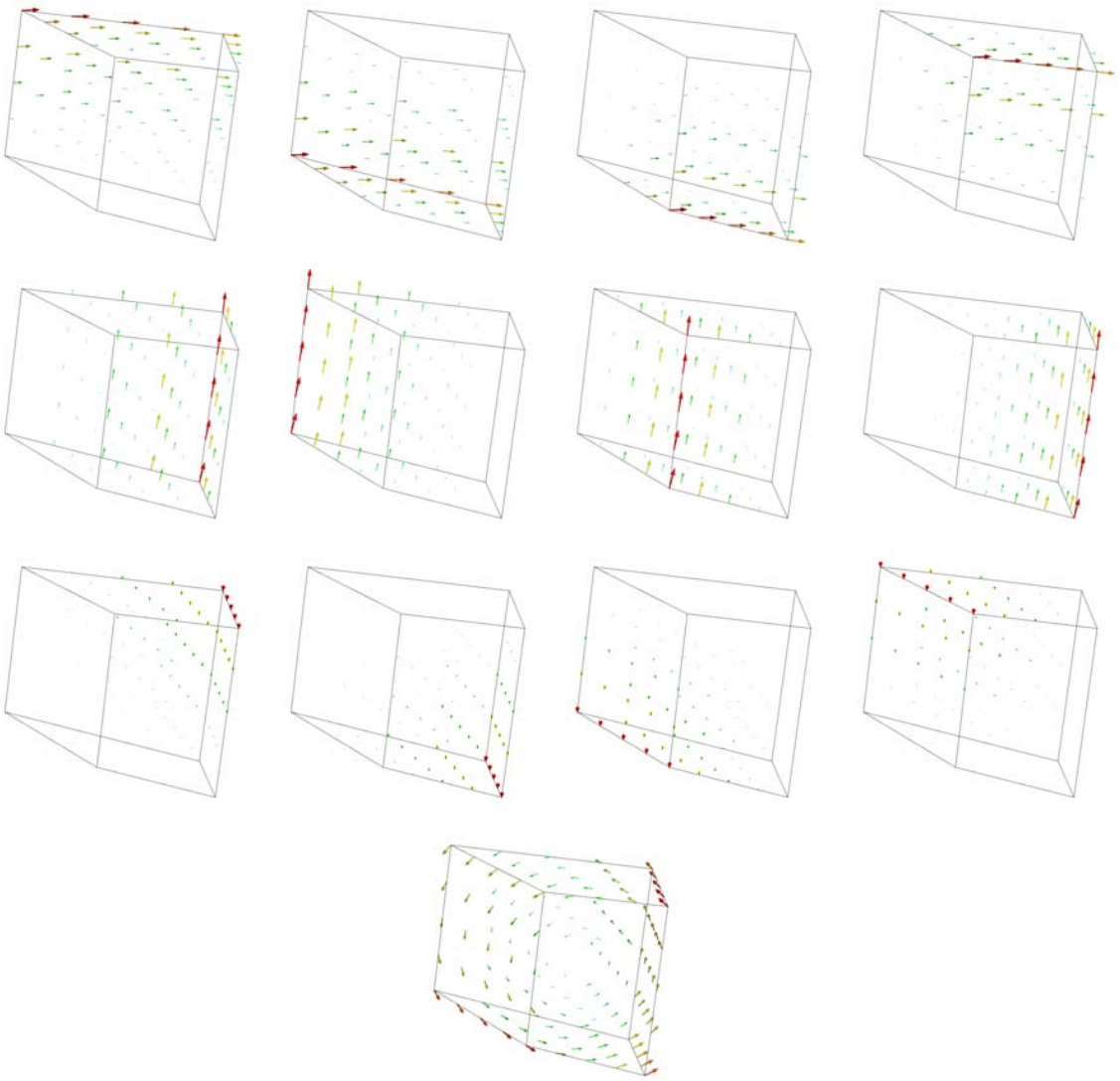


Figure 1. Edge circulation basis functions $W_{ij}^{\alpha\beta}$ (top three rows) and a weighted linear combination used to produce a vector field on a hexahedron (bottom).

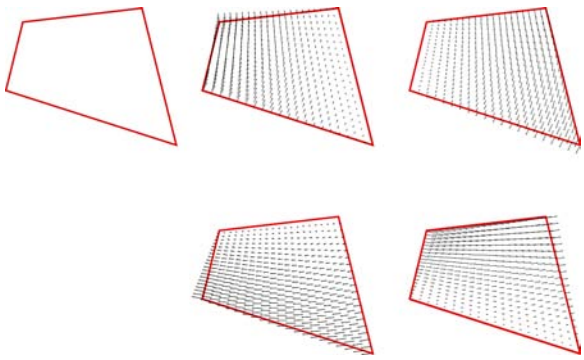


Figure 2. An illustration of edge circulation basis functions for a quadrilateral.

across their shared boundary. This implies that when an element Ω^a refers to some Γ_e, it must indicate the orientation of its edge e relative to the storage orientation.

Similarly, faces from multiple elements Ω^a and Ω^b must store not only a reference to the shared value Φ_f of the vector field on their common boundary, but also the orientation of that face relative to the orientation in which Φ_f is stored.

1.3 Simplicial and Other Element Shapes

Although not covered here, De Rham complex interpolants may also be defined over simplicial elements. Similarly, *hp*-adaptive shape functions are discussed elsewhere [6]. In general, simplicial elements are represented with $d+1$ parametric coordinates subject to the constraint that the coordinates always be a partition of unity. Reference simplicial elements are sometimes defined with one vertex at the origin and others displaced a unit distance out along mutually orthogonal axes. Other applications define reference simplicial elements as equilateral simplices with edges of length 2 [5]. Because barycentric coordinate axes are not all mutually orthogonal, sharing shape function coefficients associated with element faces is more involved than for "box" elements like quadrilaterals and hexahedra. However, as far as a storage model is concerned, simplicial shapes are no more or less difficult to represent than "box" elements.

Other novel PDE discretizations define piecewise functions over arbitrary polyhedra [3]. These are more challenging to represent and are left as future work.

The issue of how shared face fluxes and edge circulations should be stored is addressed in the next section. Following that section, the problem of how to render the stored values as well as the reconstructed vector fields is addressed. Finally, a set of API extensions and modifications are proposed that will allow the EXODUS file format to store edge and face elements.

This page intentionally left blank

2 Storing Edge and Face Element Data

This section deals with storing edge- and face-centered values in EXODUS databases as well as how that disk storage maps to Sandia's in-core visualization framework for higher order elements. As mentioned in the introduction, this is largely an exercise of mapping the discretizations to data structures by which they are represented. In the last section, several finite element spaces – $\mathscr{W}^0(K)$, $\mathscr{W}^1(K)$, $\mathscr{W}^2(K)$, and $\mathscr{W}^3(K)$ – were developed whose basis functions are associated with vertices, edges, faces, and volumes of finite elements, respectively. Thus we need to store scalar coefficients for each vertex, edge, face, and volume of the finite element discretization in order to reconstruct a field using these bases. The EXODUS API is already equipped to store these values as arrays (of dimension 1 or 2) for $\mathscr{W}^0(K)$ (vertices) and $\mathscr{W}^3(K)$ (elements) but doesn't provide for edges and faces. SHOE provides ways to store all the require coefficients and uses some more complex structures (such as free lists for element connectivity) to boot. These structures often reduce to simple arrays when a dataset is loaded or saved and so we will propose extensions to EXODUS that continue using 1- and 2-dimensional arrays for storage of all coefficients. The relationship between the disk and in-core representations is depicted in Figure 3 and discussed in detail below.

An EXODUS database stores a mesh as a set of arrays. We will denote these arrays as follows. The nodes of the mesh are stored as a set of `NP` points:

`NP`
$x_i^0, x_i^1, x_i^2, \quad i \in \{1, \ldots, NP\}$

where x_i^j are the coordinates of point i. Blocks of elements are then created, each referring back to this set of points. For example, the connectivity of a block of `NQ1` quadrilaterals would be:

`NQ1`
$p_0^j, \ldots, p_3^j, \quad j \in \{1, \ldots, NQ1\}$

where p_i^j is the integer offset into the node array of the i-th point of quadrilateral j. Similarly, the connectivity of a block of `NH1` hexahedra is stored as:

`NH1`
$p_0^j, \ldots, p_7^j, \quad j \in \{1, \ldots, NH1\}$

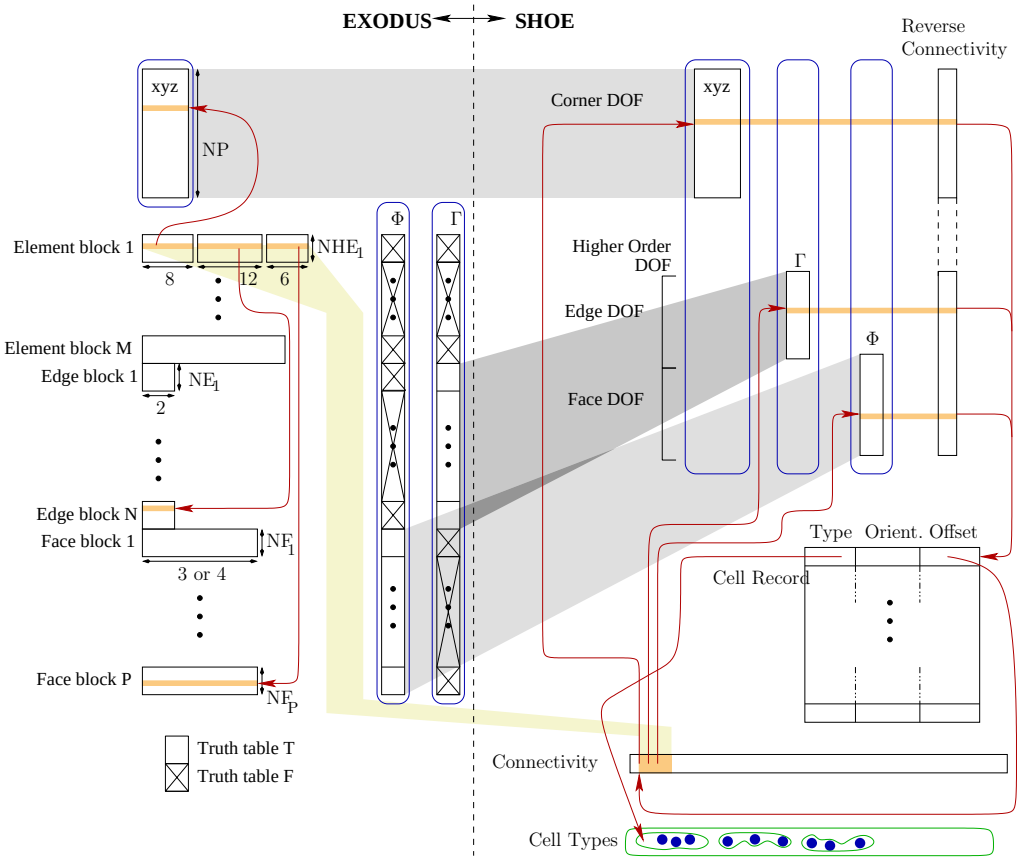


Figure 3. EXODUS and SHOE storage models (Γ: edge circulation, Φ: face fluxes). The shaded polygons show how arrays (grey) or entries of arrays (yellow) map from one model to the other. The red lines show where entries in one array reference entries in another array. Attributes of a mesh are outlined in blue.

The arrays of nodal coordinates and element block connectivities define the geometry of the mesh. Traditional EXODUS databases then allow time-varying sequences of field values defined over either nodes or cells to be added. Since our vector fields are defined by values corresponding not to cells but to lower-dimensional boundaries of those cells, we will add edge and face blocks to EXODUS files and refer those edges and faces in "extended" connectivity arrays. An edge block will consist of connectivity entries for each edge endpoint. A variable number of points per edge is allowed to handle spline and higher-order edges to be defined, but ALEGRA will create edge blocks with 2 entries per edge:

`NE1`
$p_0^j, p_1^j, \quad j \in \{1, \ldots, NE1\}$

Face blocks will consist of connectivity entries for each face corner point. ALEGRA will use 3 or 4 points per face:

NF1
$p_0^j, \ldots, p_3^j, \quad j \in \{1, \ldots, NF1\}$

Thus edge and face blocks specify line segments and quadrilateral (or triangular) faces that form the boundary of 2-D and/or 3-D finite elements. When an element block contains elements with edge circulations or face fluxes, each 2-D or 3-D finite element's connectivity may then be extended by passing optional edge connectivity and/or face connectivity arrays. The edge and face connectivity information is stored in arrays separate from nodal connectivity to maintain backward-compatibility with software using the current EXODUS API. These programs would be able to display meshes with edge and face blocks but would not be aware of attributes defined over edges and faces. The optional arrays define edges and faces of the element as offsets into the the file-local ordering of all edges in all edge-blocks or faces in all face blocks. For example, a block of extended hexahedra would be defined by these three arrays:

NHE1
$p_0^j, \ldots, p_7^j, \quad j \in \{1, \ldots, NHE1\}$
NHE1
$e_0^j, \ldots, e_{11}^j, \quad j \in \{1, \ldots, NHE1\}$
NHE1
$f_0^j, \ldots, f_5^j, \quad j \in \{1, \ldots, NHE1\}$

where each e_i^j is the integer ID of an edge in the local file order of the edges defined in all edge blocks and each f_i^j is the integer ID of a face in the local file order of the faces defined in all face blocks. Similarly, extended quadrilaterals are stored with references to edges defined in edge blocks:

NQE1
$p_0^j, \ldots, p_3^j, \quad j \in \{1, \ldots, NQE1\}$
NQE1
$e_0^j, \ldots, e_3^j, \quad j \in \{1, \ldots, NQE1\}.$

Figure 4 shows the edge ordering that should be used by ALEGRA to enumerate the edges of a hexahedral element. Hexahedral face order and quadrilateral edge order must match the existing EXODUS orders specified in the EXODUS manual.

An attribute (i.e., a set of scalar or vector field values) can then be associated with the nodes, edges, faces, volumes, or any combination thereof by storing values for each point,

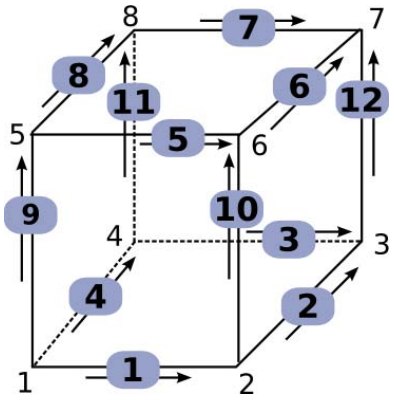


Figure 4. Edge numbers for specifying edges in the side set of a hexahedral element.

edge in an edge block, face in a face block, element in an element block, or any combination of those. The extensions to the EXODUS API that follow this section make a distinction between element blocks, face blocks, and edge blocks; although they all contain the same type of information (integer offsets into global arrays of element boundaries), they serve different purposes. Currently, the EXODUS database contains a truth table specifying which element blocks define values for a given attribute. We are proposing that edge and face blocks be included in the same truth table so that attributes may be defined over edge blocks alone (for edge circulations), face blocks alone (for face fluxes), or some combination of element, face, and edge blocks (for higher order finite elements – this capability will not necessarily be used by ALEGRA).

EXODUS API extensions are also included that generalize the notion of a side set to include sets of edges and faces and (for completeness) elements. These are intended for use in specifying boundary conditions on subsets of edges and faces in element blocks. In the case of element sets, they are intended for use mainly by post-processing tools to identify sets of elements that define features or that have been selected by a person for further examination. Edge and face sets are more like side sets than node sets because they require an integer offset into the list of all edges or faces in a file plus a bit of information to denote the orientation of the edge or face relative to storage order. Element sets are most like node sets because they do not require any extra information beyond a number that identifies a set entry.

2.1 Storage Order

Where EXODUS has many element, face, and edge blocks for connectivity, SHOE has a single linear connectivity array. However, SHOE also contains an array of records for each element that store (1) an offset into the connectivity array for the element, (2) the orientation of edge and face DOF with respect to storage, and (3) the type of the element (an

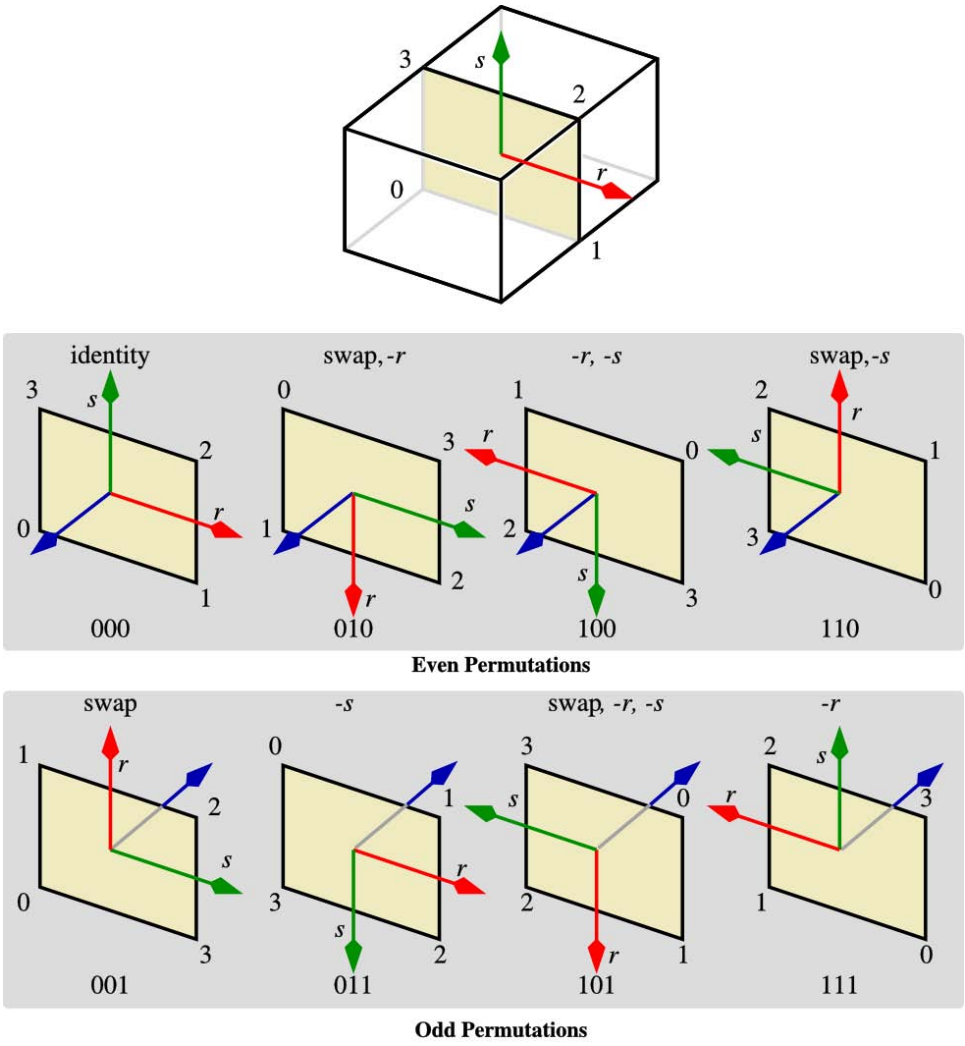


Figure 5. Possible orientations for a SHOE face element.

offset into an array of type objects that specify how that element should perform interpolation using the DOF). For edge and face elements, EXODUS explicitly denotes the storage orientation with the connectivity in each edge block and face block. SHOE can determine how an edge or face of an element is oriented by examining the order of the element's nodes on that edge or face compared to the the order they are specified in the the edge or face block. Edges can be forwards or reversed with respect to storage. Faces, on the other hand, have a total of 8 possible permutations of their nodes with respect to storage order (see Figure 5). As far as ALEGRA is concerned it only matters whether the permutation is even or odd, since that determines the direction of the face normal. In general, however, all 8 permutations are distinct and SHOE stores a permutation for each face. This is necessary for *hp*-adaptive finite elements where the face-local coordinate system determines the order in which coefficients of shape functions for each face are stored.

2.2 Element Variables

In the parlance of EXODUS, a variable is a time-varying number stored on a per-node, -edge, -face, or -element basis. However, in VTK, these are known as attributes and this section will use "attribute" to refer to fields defined on the mesh. The geometric map from a reference element to world coordinates may even be considered an attribute, and SHOE treats it as any other field (attributes are shown as blue rounded boxes in Figure 3). EXODUS, however, treats geometry as a special case. EXODUS has a single, global array of nodal coordinates and then separate time-varying arrays for other fields (shown as Γ and Φ in Figure 3). In SHOE, each attribute is stored as two arrays: one for nodal values and a second that stores a combined list of edge, face, and element values. Figure 3 shows 3 SHOE attributes: the geometric map (Ξ), Γ, and Φ. The nodal array is fixed in width while the second array can have a varying number of values (DOF) per row. Each row of the second array stores values for a single edge, face, or element of the mesh. The type stored with each element contains a function pointer for each attribute that can be used to interpolate that attribute to any point interior to the element. When reading an EXODUS mesh, connectivity entries for elements that refer to faces can be converted to SHOE offsets by adding the total number of edges in the database to each face ID and subtracting one (to account for zero-based indexing in SHOE). Subtracting one from an EXODUS edge ID yields a SHOE edge ID (to account for zero-based indexing in SHOE).

2.3 Parallel Meshes

When a mesh is stored in parallel on a distributed-memory machine, any edge or face elements on processor id P_m that are shared with processors $P_n : n \in N$ should be written by all processors, but marked as ghost data unless $m < \min_N n$.

3 Rendering Edge and Face Elements

While edge circulations and face fluxes are stored as scalar numbers, they are used to reconstruct *vector* fields. Unfortunately, vector visualization of three-dimensional datasets is not a simple task because traditional glyphs used to show vector direction and magnitude often obscure much of the data. Rather than presenting a single rendering technique, we outline a set of methods for rendering edge and face elements. First, we present a set of methods aimed at rendering the raw Γ^i and Φ^j coefficients stored at each edge or face. Then we present methods to allow interpolation of the vector field to any point in the mesh.

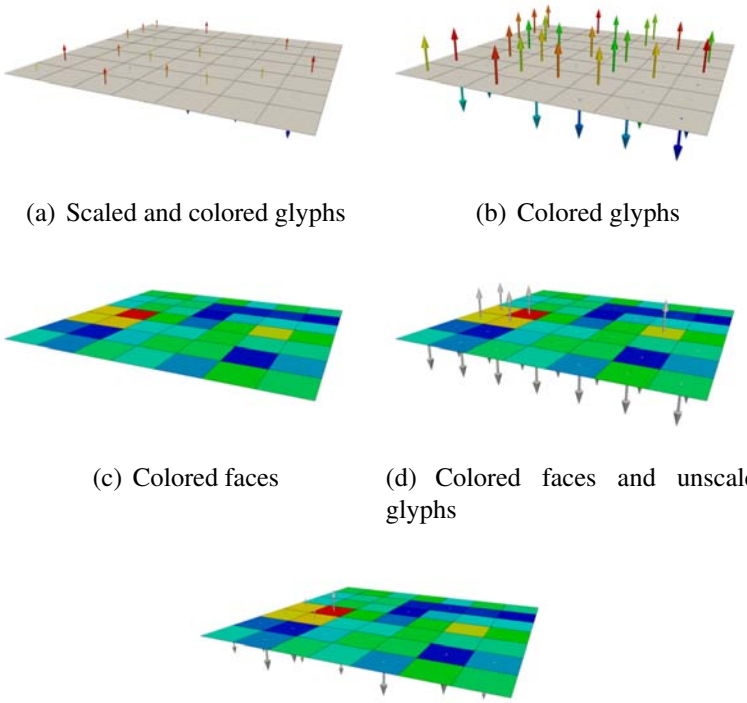
3.1 Glyphs and Color

By using colored glyphs placed along edges or faces, we can display the raw Γ^i and Φ^j coefficients. Combining colors and glyphs in different ways presents several different ways to view the same data. Glyphs can be used to show only the direction (the *sense*) or both the direction and magnitude of of an edge circulation or flux. Colors can depict the signed or the unsigned magnitude of an edge circulation or flux. Thus there are a total of 4 different renderings for edge circulations and 4 for face fluxes. Note that if magnitudes vary widely over a mesh, drawing glyphs sized proportional to coefficient magnitudes will result in either large glyphs that obscure parts of the mesh or small glyphs that are not visible. Thus glyphs should most probably be scaled to the lengths of edges or areas of faces rather than the size of coefficients. Another problem with these techniques is using color to indicate the sign of a flux. Although it may be possible to render a signed magnitude for edges, this would have no meaning without some way to show how any given edge is oriented with respect to the storage order. As an example, Figure 6 shows several possible combinations of glyphs and colorings for face fluxes.

Since these techniques do not perform any interpolation of the vector field over an element, they can be immediately realized in ParaView once the EXODUS reader is extended to allow side sets of edges in three dimensions and will work without any modifications for two-dimensional datasets.

3.2 Interpolation

Interpolation requires a a description of basis functions for each element type. ParaView does not currently support the edge or face element bases used by ALEGRA. However, VTK does provide a way for finite element packages to specify their elements as part of a "generic" dataset. Sandia has an implementation of this programming interface developed for *p*-adaptive finite elements known as SHOE (Sandia Higher Order Elements). Although ALEGRA's edge and face elements are not currently higher order, SHOE can provide lower-order basis functions for elements as well.



(a) Scaled and colored glyphs

(b) Colored glyphs

(c) Colored faces

(d) Colored faces and unscaled glyphs

(e) Colored faces and scaled glyphs

Figure 6. Combinations of color, glyph scale, and glyph magnitude mapped to face fluxes.

4 Changes to the EXODUS API

This section documents the changes that have been made to the EXODUS API as of version 4.46 (release 11/28/2006) to accomodate more general finite element discretizations, and specifically fields defined on edges and faces. We will frequently refer to the existing EXODUS documentation. Unless specifically mentioned, we refer to the March 21, 2006 draft version of the SAND92-2137 report [4] titled "EXODUS II: A Finite Element Data Model." This section is formatted similarly to the that report with the idea that it may one day become part of a later revision. To prevent confusion between elements in an EXODUS database and their associated boundaries (edges and faces which are boundaries of elements but not intended to serve as elements in their own right), we will refer to edge elements and face elements when an edge or face should be treated as a finite element in its own right and speak of edges or faces as boundaries of other, higher-dimensional finite elements. So, we use the term "edge blocks" rather than "edge element blocks," since edge blocks are meant to store boundaries of higher-dimensional finite elements.

Before proceeding, let us briefly recall some terminology used in the EXODUS manual. A *block* is an integer array of offsets into nodal coordinates. A *set* is an integer array of offsets into internal element, face, edge, or node IDs (see [4] Section 4.5 for details). An *object* refers to an array of values describing a block or set in a netCDF file: nodes, node sets, edge blocks, edge sets, face blocks, face sets, element blocks, side sets, and element sets are all objects. A *property* is an integer number assigned to each object. An *attribute* is a floating-point number assigned to each entry (node, edge, face, or element) of an object. A *variable* is a time-sequence of floating-point numbers assigned to each node, edge, face, or element in a block or set. Note that there is only one block of nodes, so all nodes must have a value for all nodal variables. Node sets may be used to define variables on a subset of nodes. Files may also have variables known as global variables.

This section adds the following types of storage to the EXODUS model:

Edge blocks Blocks defining edge endpoints.

Edge connectivity An extra connectivity array that can be stored with an element block defining a relationship between element edges and edges stored in edge blocks.

Edge sets Sets of edges over which time-constant variables (such as boundary conditions) may be defined.

Edge set distribution factors Arbitrary length, time-constant arrays associated with each edge set.

Face blocks Blocks defining faces by their corner nodes.

Face connectivity An extra connectivity array that can be stored with an element block defining a relationship between element faces and faces stored in face blocks.

Face sets Sets of faces over which time-constant variables (such as boundary conditions) may be defined.

Face set distribution factors Arbitrary length, time-constant arrays associated with each face set.

Element sets Sets of elements. These are useful for post-processing tools such as feature trackers which store subsets of elements that form features. They also allow time-constant variables to be defined over the set; this is not used by feature trackers but could be useful for imposing element-centered source/sink terms such as body forces.

Element set distribution factors Arbitrary length, time-constant arrays associated with each element set.

The changes proposed in this section require several new dimensions and arrays in the EXODUS netCDF file. Particularly, the following dimensions must be added to those listed in Appendix A of SAND92-2137:

`num_edge`	number of edges
`num_ed_blk`	number of edge blocks
`num_ed_in_blk#`	number of edges in edge block #
`num_nod_per_ed#`	number of nodes per edge in edge block #
`num_edg_per_el#`	number of edges per element in element block #
`num_att_in_eblk#`	number of attributes per edge in edge block #
`num_edge_sets`	number of edge sets
`num_edge_es#`	number of edges in edge set #
`num_df_es#`	number of distribution factors in edge set #
`num_edge_var`	number of edge variables
`num_es_var`	number of edge set variables
`num_edge_maps`	number of edge maps
`num_face`	number of faces
`num_fa_blk`	number of face blocks
`num_fa_in_blk#`	number of faces in face block #
`num_nod_per_fa#`	number of nodes per face in face block #
`num_fac_per_el#`	number of faces per element in element block #
`num_att_in_fblk#`	number of attributes per face in face block #
`num_face_sets`	number of face sets
`num_face_fs#`	number of faces in face set #
`num_df_fs#`	number of distribution factors in face set #
`num_face_var`	number of face variables
`num_fs_var`	number of face set variables
`num_face_maps`	number of face maps
`num_elem_sets`	number of element sets
`num_ele_els#`	number of elements in element set #
`num_df_els#`	number of distribution factors in element set #
`num_els_var`	number of element set variables

The following variables must be added to the underlying storage model to handle edge blocks, edge sets, face blocks, face sets, and element sets:

Type	Name/Description	Size
integer	`edgconn#`	`(num_el_in_blk#, num_edg_per_el#)`
	edge connectivity (file-local edge offsets) for element block #	
integer	`ebconn#`	`(num_ed_in_eblk#, num_nod_per_ed#)`
	edge block connectivity (node numbers) for edge block #	
integer	`ed_status`	`(num_el_blk)`
	edge block status (does `ebconn#` exist?)	
real	`eattrib#`	`(num_ed_in_blk#, num_att_in_eblk#)`
	list of attributes for edge block #	
integer	`eattrib_name#`	`(num_att_in_eblk#, len_string)`
	list of attribute names for edge block #	
integer	`ed_prop#`	`(num_ed_blk)`
	list of the #th property for all edge blocks	
integer	`ed_prop1`	`(num_ed_blk)`
	edge block IDs property (always the first edge block property)	
integer	`es_status`	`(num_edge_sets)`
	edge set status (do `edge_es#` and `ornt_es` exist?)	
real	`dist_fact_es#`	`(num_df_es#)`
	distribution factors edge set #	
integer	`edge_es#`	`(num_edge_es#)`
	list of edges in edge set #	
integer	`ornt_es#`	`(num_edge_es#)`
	array of edge orientations in edge set #	
integer	`es_prop#`	`(num_edge_sets)`
	list of the #th property for all edge sets	
integer	`es_prop1`	`(num_edge_sets)`
	edge set id property (always first edge set prop)	
integer	`edge_var_tab`	`(num_ed_blk, num_edge_var)`
	truth table describing which edge blocks store which variables	
integer	`eset_var_tab`	`(num_ed_blk, num_es_var)`
	truth table describing which edge sets store which variables	
integer	`name_edge_var`	`(num_edge_var, len_string)`
	names of edge variables	
integer	`name_eset_var`	`(num_edge_var, len_string)`
	names of edge set variables	
real	`vals_edge_var#1eb#2`	`(time_step, num_ed_in_blk#2)`
	values of edge variable #1 in edge block #2	
real	`vals_eset_var#1es#2`	`(num_edge_es#2)`
	values of edge variable #1 in edge set #2	

integer `ed_names` `(num_ed_blk, len_string)`
: names of edge blocks

integer `es_names` `(num_edge_sets, len_string)`
: names of edge sets

integer `edmap_names` `(num_edge_maps, len_string)`
: names of edge maps

integer `edge_map#` `(num_edge)`
: array of edge IDs in edge map #

integer `edm_prop#` `(num_edge_maps)`
: array of the #-th property for each edge map

---

integer `facconn#` `(num_el_in_blk#, num_fac_per_el#)`
: face connectivity (file-local face offsets) for element block #

integer `fbconn#` `(num_fa_in_blk#, num_nod_per_fa#)`
: face block connectivity (node IDs) for face block #

integer `fa_status` `(num_el_blk)`
: face block connectivity status (does `fbconn#` exist?)

real `fattrib#` `(num_fa_in_blk#, num_att_in_fblk#)`
: list of attributes for face block #

integer `fattrib_name#` `(num_att_in_fblk#, len_string)`
: list of attribute names for face block #

integer `fa_prop#` `(num_fa_blk)`
: list of the #th property for all face blocks

integer `fa_prop1` `(num_fa_blk)`
: face block IDs (always the first face block property)

integer `fs_status` `(num_face_sets)`
: face set status (do `face_fs#` and `ornt_fs` exist?)

real `dist_fact_fs#` `(num_df_fs#)`
: distribution factors for face set #

integer `face_fs#` `(num_face_fs#)`
: list of faces in face set #

integer `ornt_fs#` `(num_face_fc#)`
: array of face orientations in face set #

integer `fs_prop#` `(num_face_sets)`
: list of the #th property for all face sets

integer `fs_prop1` `(num_face_sets)`
: face set id property (always first face set prop)

integer `face_var_tab` `(num_fa_blk, num_face_var)`
: truth table describing which face blocks store which variables

integer `fset_var_tab` `(num_fa_blk, num_fs_var)`
: truth table describing which face sets store which variables

integer `name_face_var` `(num_face_var, len_string)`
: names of face variables

integer `name_fset_var` `(num_edge_var, len_string)`
: names of face set variables

real `vals_face_var#1fb#2 (time_step, num_fa_in_blk#2)`
values of face variable #1 in face block #2

real `vals_fset_var#1fs#2 (num_face_fs#2)`
values of face variable #1 in face set #2

integer `fa_names (num_fa_blk, len_string)`
names of face blocks

integer `fs_names (num_face_sets, len_string)`
names of face sets

integer `famap_names (num_face_maps, len_string)`
names of face maps

integer `face_map# (num_face)`
array of face IDs in face map #

integer `fam_prop# (num_face_maps)`
array of the #-th property for each face map

---

integer `els_status (num_elem_sets)`
elem set status (does `elem_els#` exist?)

real `dist_fact_els# (num_df_els#)`
distribution factors for elem set #

integer `elem_els# (num_ele_els#)`
list of elems in elem set #

integer `els_prop# (num_elem_sets)`
list of the #th property for all elem sets

integer `els_prop1 (num_elem_sets)`
elem set id property (always first elem set prop)

integer `elset_var_tab (num_fa_blk, num_els_var)`
truth table describing which elem sets store which variables

integer `name_elset_var (num_els_var, len_string)`
names of element set variables

real `vals_elset_var#1es#2(num_ele_els#2)`
values of element set variable #1 in element set #2

integer `els_names (num_elem_sets, len_string)`
names of elem sets

4.0.1 Shape vs. Interpolant

This section discusses some difficulties with the current use of the `elem_type` attribute. It should be pointed out that unlike the rest of this report, the suggestions in this section are **not implemented** in EXODUS 4.46. They are intended for future versions of the EXODUS API.

Finally, some new attributes are defined in place of the existing `elem_type` attribute which should be deprecated. The `elem_type` attribute should be replaced because it does not

allow distinctions between the shape of an element and the way in which values are interpolated over the element. For example, `HEX8` and `HEX20` are common values for `elem_type` but they conflate the shape (`HEX`) with the interpolant family (Lagrange and serendipity, respectively) and order ($[1,1,1]$ and $[2,2,2]$, respectively). Also, they assume that all result variables will employ the same interpolant over all elements in a given block. This need not be true. Therefore, `elem_type` should be replaced with:

Type	Name	Description
string	`elem_shape`	The shape of elements in a given element block. This is similar to the `elem_type` attribute but without the number of nodes appended to the shape. Attached to the `connect` variable.
string	`elem_interp`	The name of interpolant function family for each result variable + element block combination. Attached to the `vals_elem_var#1eb#2` variable
string	`elem_orders`	A comma-separated list of polynomial order(s) for each result variable + element block combination. Attached to the `vals_elem_var#1eb#2` variable.

4.1 Data File Utilities

4.1.1 Write Initialization Parameters

The functions `ex_put_init` and `ex_put_init_ext` write initialization parameters to the EXODUS II file. Exactly one of these functions must be called once (and only once) before writing any data to the file. The difference between the two functions is that the former, while backwards-compatible, will produce a database incapable of storing edge and face blocks and edge, face, and element sets. Note that you may still create element blocks that contain edges or faces, but `ex_put_init` precludes element blocks where 2-D and 3-D elements refer to 1-D bounding edges and 2-D bounding faces in special edge and face blocks as discussed in the first part of this document. If you require this functionality (i.e., you would like to store attributes with values specified on edges or faces rather than the nodes of an element), then you should call `ex_put_init_ext` instead.

Rather than require a large number of arguments, this function takes the address of a C structure as its input. The structure is defined as

ex_init_params: C Structure

```
struct ex_init_params {
    char title[MAX_LINE_LENGTH + 1]
        Database title. The maximum length is MAX_LINE_LENGTH.
    int num_dim
        The dimension of point coordinates. This should be 2 or 3.
```

`int num_nodes`
: The number of nodal points to be inserted into the database.

`int num_edge`
: The number of edge elements to be inserted into the database.

`int num_edge_blk`
: The number of edge blocks. The sum of the number of edges in each edge block over all edge blocks must be `num_edge`.

`int num_face`
: The number of faces to be inserted into the database.

`int num_face_blk`
: The number of face blocks. The sum of the number of faces in each face block over all face blocks must be `num_face`.

`int num_elem`
: The number of elements to be inserted into the database.

`int num_elem_blk`
: The number of element blocks. The sum of the number of elements in each element block over all element blocks must be `num_elem`.

`int num_node_sets`
: The number of node sets.

`int num_edge_sets`
: The number of edge sets.

`int num_face_sets`
: The number of face sets.

`int num_side_sets`
: The number of side sets.

`int num_elem_sets`
: The number of element sets.

`int num_node_maps`
: The number of node maps.

`int num_edge_maps`
: The number of edge maps.

`int num_face_maps`
: The number of face maps.

`int num_elem_maps`
: The number of element maps.

}

In case of an error, `ex_put_init_ext` returns a negative number; a warning will return a positive number. Zero is returned on success. Possible causes of errors include:

- data file not properly opened with a called to `ex_create` or `ex_open`.
- data file opened for read only.
- this routine has been called previously.

Previously, `ex_put_init` did not take arguments for the number of edges and faces, the number of edge and face blocks, the number of edge and face sets, and the number of maps

Note

of any type. We work around this by adding a new call: ex_put_init_ext that duplicates the functionality of ex_put_init and also takes these additional arguments. This means that only one of ex_put_init and ex_put_init_ext should ever be called for a given database. All documentation for API routines that note ex_put_init must be called prior to their execution should now note that either ex_put_init or ex_put_init_ext should be called.

ex_put_init_ext: C Interface

```
int ex_put_init_ext(exoid, ex_params);
```

`int exoid` **(R)**

EXODUS file ID returned from a previous call to ex_create or ex_open.

`const ex_init_params* ex_params` **(R)**

The model parameters used to initialize the EXODUS database. See the description of ex_init_params above for details.

4.1.2 Read Initialization Parameters

The function ex_get_init_ext reads initialization parameters from an opened EXODUS II file. It is an extension of ex_get_init that also returns information about edge blocks, face blocks, edge sets, face sets, and element sets. Unless your application will not make use of this information, you should use ex_get_init_ext instead of ex_get_init.

Rather than require a large number of arguments, this function takes the address of a C structure as its input. The ex_init_params structure is defined in §4.1.1.

In case of an error, ex_get_init_ext returns a negative number; a warning will return a positive number. Zero is returned on success. Possible causes of errors include:

- data file not properly opened with a called to ex_create or ex_open.

Note

Previously, ex_get_init did not take arguments for the number of edges and faces and the number of edge and face blocks. A possible workaround to changing the API is to require a two-step initialization process where the original ex_get_init would be called followed by an optional call to ex_get_init_edge_face_elem, which would take values for the four new parameters. Another possible workaround (and the one chosen for implementation) is to define two alternative calls that perform the same task: the original ex_get_init and the new ex_get_init. Since C does not allow overloading, the new ex_get_init is named ex_get_init_ext.

ex_get_init_ext: C Interface

```
int ex_get_init_ext(exoid, ex_params);
```

`int exoid` **(R)**
: EXODUS file ID returned from a previous call to ex_create or ex_open.

`ex_init_params* ex_params` **(W)**
: The model parameters describing the number of blocks and sets in the EXODUS database. See §4.1.1 for details on the structure.

4.1.3 Modified! Inquire EXODUS Parameters

Note

The ex_inquire function must be extended to include several new parameters that deal with edges and faces. The req_info argument must now also accept the following additional values:

EX_INQ_EDGE	The number of edges (defined across all edge blocks) is returned in ret_int.
EX_INQ_EDGE_BLK	The number of edge blocks is returned in ret_int.
EX_INQ_EDGE_SETS	The number of edge sets is returned in ret_int.
EX_INQ_ES_LEN	The length of the concatenated edge set edge list is returned in ret_int.
EX_INQ_ES_DF_LEN	The length of the concatenated edge set distribution factor list is returned in ret_int.
EX_INQ_EDGE_PROP	The number of integer properties stored for each edge block is returned in ret_int; this includes the properties named "ID".
EX_INQ_ES_PROP	The number of integer properties stored for each edge set is returned in ret_int.
EX_INQ_FACE	The number of faces (defined across all face blocks) is returned in ret_int.
EX_INQ_FACE_BLK	The number of face blocks is returned in ret_int.
EX_INQ_FACE_SETS	The number of face sets is returned in ret_int.
EX_INQ_FS_LEN	The length of the concatenated face set face list is returned in ret_int.
EX_INQ_FS_DF_LEN	The length of the concatenated face set distribution factor list is returned in ret_int.
EX_INQ_FACE_PROP	The number of integer properties stored for each face block is returned in ret_int; this includes the properties named "ID".
EX_INQ_FS_PROP	The number of integer properties stored for each face set is returned in ret_int.
EX_INQ_ELEM_SETS	The number of element sets is returned in ret_int.
EX_INQ_ELS_LEN	The length of the concatenated element set element list is returned in ret_int.
X_INQ_ELS_DF_LEN	The length of the concatenated element set distribution factor list is returned in ret_int.
EX_INQ_ELS_PROP	The number of integer properties stored for each element set is returned in ret_int.

4.2 Model Description

4.2.1 Write Node, Edge, Face, or Element Number Map

The function ex_put_num_map writes out an optional number map to the database. The value of map_type determines whether the number map entries are associated with nodes (EX_NODE_MAP), edges (EX_EDGE_MAP), faces (EX_FACE_MAP), or elements (EX_ELEMENT_MAP). Either ex_put_init_ext or ex_put_init must be invoked before this call is made. The function ex_put_init_ext must be invoked before this call is made if map_type is EX_EDGE_MAP or EX_FACE_MAP.

For efficient writes, you should call ex_put_concat_all_blocks with define_maps set to a nonzero number before calling ex_put_num_map. Calling ex_put_concat_all_blocks in this way allocates storage in the file for all of the maps at once, rather than one at a time, which can become expensive.

In case of an error, ex_put_num_map returns a negative number; a warning will return a positive number. Zero is returned on success. Possible causes of errors include:

- data file not properly opened with call to ex_create or ex_open.
- data file opened for read only.
- data file not initialized properly with a call to ex_put_init_ext or ex_put_init.
- an number map for the given map_type alrady exists in the file.

ex_put_num_map: C Interface

```
int ex_put_num_map(exoid, map_type, map_id, map);
```

`int exoid` **(R)**
: EXODUS file ID returned from a previous call to ex_create or ex_open.

`int map_type` **(R)**
: One of EX_NODE_MAP, EX_EDGE_MAP, EX_FACE_MAP, or EX_ELEMENT_MAP.

`int map_id` **(R)**
: A unique (among all maps of the same type) identifier for the map.

`const int* map` **(R)**
: The number map. There must be an entry in map for every node, edge, face, or element (depending on the value of map_type) in the database (across all blocks, where applicable).

4.2.2 Read Number Map

The function ex_get_num_map reads a number map for entries of the given map_type from the database. The value of map_type determines whether the number map entries are associated with nodes (EX_NODE_MAP), edges (EX_EDGE_MAP), faces (EX_FACE_MAP), or elements (EX_ELEMENT_MAP). If a number map is not stored in the data file, an error is returned. Unlike other EXODUS calls dealing with maps, no default array is assumed since the purpose of each map is left to the application writing the map. Memory must be allocated for the map array (num_nodes, num_edge, num_face, or num_elem in length) before this call is made.

In case of an error, ex_get_num_map returns a negative number; a warning will return a positive number. Zero is returned on success. Possible causes of errors include:

- data file not properly opened with a call to ex_create or ex_open.
- if a number map is not stored, an error is returned. No default map is assumed.

ex_get_num_map: C Interface

```
int ex_get_num_map(exoid, map_type, map_id, map);
```

`int exoid` **(R)**
EXODUS file ID returned from a previous call to ex_create or ex_open.

`int map_type` **(R)**
One of EX_NODE_MAP, EX_EDGE_MAP, EX_FACE_MAP, or EX_ELEMENT_MAP.

`int map_id` **(R)**
The ID of the map. Use ex_get_ids to retrieve a list of all map ids for a given map_type.

`int* map` **(W)**
The returned edge number map.

4.2.3 Write Edge Order Map

The `ex_put_map_ext` function is not currently necessary. Although it may be implemented in the future, it need not be implemented for initial edge and face support.

4.2.4 Read Edge Order Map

The `ex_get_map_ext` function is not currently necessary. Although it may be implemented in the future, it need not be implemented for initial edge and face support.

4.2.5 Write Edge, Face, or Element Block Parameters

The function ex_put_block writes the parameters used to describe a block. The particular type of block (edge, face, or element) is specified by passing EX_EDGE_BLOCK, EX_FACE_BLOCK, or EX_ELEM_BLOCK in the blk_type argument. Note that num_edges_per_entry and num_faces_per_entry are not used unless blk_type is EX_ELEM_BLOCK. If this is called for an element block that does not have optional edge and/or face extended connectivity arrays, simply pass 0 or -1 for num_edges_per_entry and/or num_faces_per_entry, respectively.

In case of an error, ex_put_block will return a negative number; a warning will return a positive number. Zero is returned on success. Possible causes of errors include:

- data file not properly opened with a call to ex_create or ex_open.
- data file opened for read only
- data file not initialized properly with a call to ex_put_init_ext. Note that calling ex_put_init is not sufficient unless you limit blk_type to ELEMENT_BLOCK.
- a block with the same type and ID has already been specified.
- the number of edge blocks specified in the call to ex_put_init_ext has been exceeded.

ex_put_block: C Interface

```
int ex_put_block(exoid, blk_type, blk_id, blk_type_name,
num_entries_this_blk, num_nodes_per_entry, num_edges_per_entry,
num_faces_per_entry, num_attr_per_entry);
```

`int exoid` **(R)**
: EXODUS file ID returned from a previous call to ex_create or ex_open.

`int blk_type` **(R)**
: One of EX_EDGE_BLOCK, EX_FACE_BLOCK, or EX_ELEM_BLOCK.

`int blk_id` **(R)**
: The block ID.

`const char* blk_type_name` **(R)**
: A string describing the type of entries in this block. The maximum length of this string is MAX_STR_LENGTH. When blk_type is EX_EDGE_BLOCK, the only valid value for this string is STRAIGHT. In the future other values may be used for curved edges (such as LAGRANGE2 for quadratic, Lagrange-interpolated edges). When blk_type is EX_FACE_BLOCK, this string should be TRIANGLE or QUADRILATERAL. When blk_type is EX_ELEM_BLOCK, this string can be the name of any valid EXODUS element shape.

`int num_entries_this_blk` **(R)**
: The number of entries in the edge block.

`int num_nodes_per_entry` **(R)**

The number of nodes per entry in the block. This should be 2 for all `STRAIGHT` edges, 3 for `TRI3`, 4 for `QUAD4`, etc.

`int num_edges_per_entry` **(R)**

This value is ignored unless `blk_type` is `EX_ELEM_BLOCK`. The number of edges per element in the block. This should depend solely on the shape of the element (and not the number of nodes associated with the element). For example, all hexahedra (HEX8, HEX20, HEX27) should specify 12 edges per element. If this function is invoked for an element block that is not extended, use a value of 0 or -1.

`int num_faces_per_entry` **(R)**

This value is ignored unless `blk_type` is `EX_ELEM_BLOCK`. The number of faces per element in the block. This should depend solely on the shape of the element (and not the number of nodes associated with the element). For example, all hexahedra (HEX8, HEX20, HEX27) should specify 6 faces per element. If this function is invoked for an element block that is not extended, use a value of 0 or -1.

`int num_attr_per_entry` **(R)**

The number of attributes per entry in the block.

4.2.6 Read Edge, Face, or Element Block Parameters

The function ex_get_block reads the parameters used to describe a block. The particular type of block (edge, face, or element) must be specified by passing EX_EDGE_BLOCK, EX_FACE_BLOCK, or EX_ELEM_BLOCK in the blk_type argument; the blk_type argument is not a returned value. Note that num_edges_per_entry and num_faces_per_entry are not used unless blk_type is EX_ELEM_BLOCK; when blk_type is EX_EDGE_BLOCK or EX_FACE_BLOCK, you may pass NULL for num_edges_per_entry and num_faces_per_entry. If this is called for an element block that does not have optional extended connectivity information, special values will be returned in num_edges_per_entry and num_faces_per_entry.

In case of an error, ex_get_block will return a negative number; a warning will return a positive number. Zero is returned on success. Possible causes of errors include:

- data file not properly opened with a call to ex_create or ex_open
- a block with the specified blk_type and ID is not stored in the data file

ex_get_block: C Interface

```
int ex_get_block(exoid, blk_type, blk_id, blk_type_name,
num_entries_this_blk, num_nodes_per_entry, num_edges_per_entry,
num_faces_per_entry, num_attr_per_entry);
```

`int exoid` **(R)**
: EXODUS file ID returned from a previous call to ex_create or ex_open.

`int blk_type` **(R)**
: One of EX_EDGE_BLOCK, EX_FACE_BLOCK, or EX_ELEM_BLOCK.

`int blk_id` **(R)**
: The block ID.

`char* blk_type_name` **(W)**
: The type of entries in this block. The maximum length of this string is MAX_STR_LENGTH.

`int* num_entries_this_blk` **(W)**
: The number of entries in the edge block.

`int* num_nodes_per_entry` **(W)**
: The number of nodes per entry in the block. This should be 2 for all STRAIGHT edges, 3 for TRI3s, 4 for QUAD4s, etc.

`int* num_edges_per_elem` **(W)**
: The number of edges per element in the block. This should depend solely on the shape of the element (and not the number of nodes associated with the element). For example, all hexahedra (HEX8, HEX20, HEX27) should specify 12 edges per element. If this function is invoked for an element block that is not extended, this will be -1.

`int* num_faces_per_elem` **(W)**

The number of faces per element in the block. This should depend solely on the shape of the element (and not the number of nodes associated with the element). For example, all hexahedra (HEX8, HEX20, HEX27) should specify 6 faces per element. If this function is invoked for an element block that is not extended, this will be -1.

`int* num_attr_per_entry` **(W)**

The number of attributes per entry in the block.

4.2.7 Write All Edge, Face, and Element Block Parameters

If edge, face, or element blocks are written using `ex_put_block` or `ex_put_elem_block`, significant inefficiencies can result because the NETCDF file is completely re-written after each call. For large files with many edge, face, or element blocks, this can be prohibitively slow. To avoid this problem, the function `ex_put_concat_all_blocks` may be used to write the block parameters for all edge, face, and element blocks in a single call. If you called `ex_put_init`, you should call `ex_put_concat_elem_block` instead of `ex_put_concat_all_blocks`.

Rather than requiring a long list of arguments, `ex_put_concat_all_blocks` takes a C structure named `ex_block_params` containing all of the information about the various blocks. The structure is defined as

ex_block_params: C Structure

```
struct ex_block_params {
```

`int* edge_blk_id`
: An array of edge block IDs.

`char** edge_type`
: The types of edges in each of the edge blocks. The maximum length of this string is `MAX_STR_LENGTH`.

`int* num_edge_this_blk`
: The number of edges in each of the edge blocks.

`int* num_nodes_per_edge`
: The number of nodes per edge in each of the edge blocks.

`int* num_attr_edge`
: The number of attributes per edge in each of the edge blocks.

`int* face_blk_id`
: An array of face block IDs.

`char** face_type`
: The types of faces in each of the face blocks. The maximum length of this string is `MAX_STR_LENGTH`.

`int* num_face_this_blk`
: The number of faces in each of the face blocks.

`int* num_nodes_per_face`
: The number of nodes per face in each of the face blocks.

`int* num_attr_face`
: The number of attributes per face in each of the face blocks.

`int* elem_blk_id`
: An array of element block IDs.

`char** elem_type`
: The types of elements in each of the element blocks. The maximum length of this string is `MAX_STR_LENGTH`.

`int* num_elem_this_blk`
: The number of elements in each of the element blocks.

`int* num_nodes_per_elem`
: The number of nodes per element in each of the element blocks.

`int* num_edges_per_elem`
: The number of edges per element in each of the element blocks. Use 0 or -1 for element blocks that do not specify edge connectivity with offsets into edge blocks.

`int* num_faces_per_elem`
: The number of faces per element in each of the element blocks. Use 0 or -1 for element blocks that do not specify face connectivity with offsets into face blocks.

`int* num_attr_elem`
: The number of attributes per element in each of the element blocks.

`int define_maps`
: Zero if `node_number_map` and `elem_number_map` will not be written later; nonzero if they will. This is just an optimization that will predefine the space for the maps now if they will be written later.

}

Note

It is unclear whether these edge-face extensions would now require all element blocks, edge blocks, and face blocks to be written in a single call to prevent the NETCDF file from being rewritten 3 times (assuming `ex_put_concat_elem_block`, `ex_put_concat_edge_block`, and `ex_put_concat_face_block` were each called once). To be on the safe side (and create a smaller API), I'm assuming that all of these should be specified at once.

In case of an error, `ex_put_concat_all_blocks` will return a negative number; a warning will return a positive number. Zero is returned on success. Possible causes of errors include:

- data file not properly opened with a call to `ex_create` or `ex_open`.
- data file opened for read only.
- data file not initialized properly with call to `ex_put_init_ext`. If you called `ex_put_init` instead, you should call `ex_put_concat_elem_block` instead of `ex_put_concat_all_blocks`.

ex_put_concat_all_blocks: C Interface

```
int ex_put_concat_all_blocks(exoid, ex_bparams);
```

`int exoid` **(R)**
: EXODUS file ID returned from a previous call to `ex_create` or `ex_open`.

`const ex_block_params* ex_bparams` **(R)**
: Information describing the number of elements in each block and their associated block IDs.

4.2.8 Modified! Read Edge, Face, or Element Block IDs or Node, Edge, Face, Side, or Element Set IDs

The function ex_get_ids reads the IDs of all the blocks or sets of the requested obj_type. The obj_type argument must be one of EX_NODE_SET, EX_EDGE_SET, EX_FACE_SET, EX_SIDE_SET, EX_ELEM_SET, EX_EDGE_BLOCK, EX_FACE_BLOCK, or EX_ELEM_BLOCK. Memory must be allocated for the returned array of (num_node_sets, num_edge_sets, num_face_sets, num_side_sets, num_elem_sets, num_edge_blk, num_face_blk, or num_elem_blk, respectively, depending on the value of obj_type) IDs before the function is invoked. The required size can be determined via a call to ex_inquire.

In case of an error, ex_get_ids will return a negative number; a warning will return a positive number. Zero is returned on success. Possible causes of errors include:

- data file not properly opened with a call to ex_create or ex_open

ex_get_ids: C Interface

```
int ex_get_ids(exoid, obj_type, obj_ids);
```

int exoid **(R)**
: EXODUS file ID returned from a previous call to ex_create or ex_open.

int obj_type **(R)**
: One of EX_NODE_SET, EX_EDGE_SET, EX_FACE_SET, EX_SIDE_SET, EX_ELEM_SET, EX_EDGE_BLOCK, EX_FACE_BLOCK, or EX_ELEM_BLOCK.

int* obj_ids **(W)**
: Returned array of the block IDs. The order of the IDs in this array reflects the sequence that the objects were introduced into the file.

The following code will read edge set ids and then the associated edge sets from an open EXODUS II file:

```
int num_edge_sets, error, exoid, num_edges_in_set, num_df_in_set,
    *ids, *edge_list, i;
float* dist_fact;

error = ex_inq( exoid, EX_INQ_EDGE_SETS, &num_edge_sets, &fdum, cdum );

ids = (int*) calloc( num_edge_sets, sizeof(int) );
error = ex_get_ids( exoid, EX_EDGE_SET, ids );

for ( i = 0; i < num_edge_sets; ++i ) {
  error = ex_get_set_param( exoid, EX_EDGE_SET, ids[i],
    &num_edges_in_set, &num_df_in_set );
```

```
  edge_list = (int*) calloc( num_edges_in_set, sizeof(int) );
  error = ex_get_set( exoid, EX_EDGE_SET, ids[i], edge_list );
  if ( num_df_in_set > 0 ) {
    dist_fac = (float*) calloc( num_df_in_set, sizeof(float) );

    error = ex_get_set_dist_fact( exoid, EX_EDGE_SET, ids[i], dist_fac );
  }
}
```

4.2.9 Write Edge, Face, or Element Block Connectivity

The function ex_put_conn writes the connectivity array for a block of the requested blk_type. The blk_type argument may be one of EX_EDGE_BLOCK, EX_FACE_BLOCK, or EX_ELEM_BLOCK. The function ex_put_block or ex_put_concat_all_blocks must be invoked before this call is made.

If blk_type is EX_EDGE_BLOCK or EX_FACE_BLOCK, elem_edge_conn and elem_face_conn are unused and NULL pointers should be passed. If blk_type is EX_ELEM_BLOCK, elem_edge_conn and elem_face_conn must point to edge and face connectivity arrays as specified by the values passed in the num_edges_per_elem and num_faces_per_elem arguments to ex_put_block or ex_put_all_concat. If 0 or -1 was specified for num_edges_per_elem or num_faces_per_elem, then NULL pointers should be passed for elem_edge_conn or elem_face_conn, respectively.

In case of an error, ex_put_conn returns a negative number; a warning will return a positive number. Zero is returned on success. Possible causes of errors include

- data file opened for read only
- data file not initialized properly with call to ex_put_init (for EX_ELEM_BLOCK) or ex_put_init_ext (all other values of blk_type).
- ex_put_block was not called previously with the same value of blk_type as passed to ex_put_conn.

ex_put_conn: C Interface

```
int ex_put_conn(exoid, blk_type, blk_id, node_conn,
elem_edge_conn, elem_face_conn);
```

`int exoid` **(R)**
: EXODUS file ID returned from a previous call to ex_create or ex_open.

`int blk_type` **(R)**
: One of EX_EDGE_BLOCK, EX_FACE_BLOCK, or EX_ELEM_BLOCK.

`int blk_id` **(R)**
: The block ID whose connectivity is being specified.

`const int node_conn[num_entries_this_blk,num_nodes_per_entries]` **(R)**
: The connectivity array; a list of nodes (internal node IDs; see section 4.5) that define each entry in the block. The node index cycles faster than the edge, face, or element index. The type of entry (edge, face, or element) depends on the values of blk_type.

`const int elem_edge_conn[num_elems_this_blk,num_edges_per_elem]` **(R)**
: The edge connectivity array; a list of edges (internal edge IDs) that define each element in the block. This should be NULL if blk_type is EX_EDGE_BLOCK or EX_FACE_BLOCK, or if num_edges_per_elem is 0 or -1. The edge index cycles faster than the element index.

`const int elem_face_conn[num_elems_this_blk,num_faces_per_elem]` **(R)**

The face connectivity array; a list of faces (internal face IDs) that define each element in the block. This should be NULL if `blk_type` is `EX_EDGE_BLOCK` or `EX_FACE_BLOCK`, or if `num_faces_per_elem` is 0 or -1. The face index cycles faster than the element index.

4.2.10 Read Edge, Face, or Element Block Connectivity

The function ex_get_conn reads the connectivity array for an edge, face, or element block. Memory must be allocated for the nodal connectivity array (num_edges_this_blk * num_nodes_per_edge, num_faces_this_blk * num_nodes_per_face, or num_elems_this_blk * num_nodes_per_elem in length) before this routine is called. An entry is an edge, face, or element when the argument blk_type is EX_EDGE_BLOCK, EX_FACE_BLOCK, or EX_ELEM_BLOCK, respectively. If blk_type is EX_EDGE_BLOCK or EX_FACE_BLOCK, elem_edge_conn and elem_face_conn are ignored and NULL pointers should be passed.

If blk_type is EX_ELEM_BLOCK, elem_edge_conn and elem_face_conn must point to enough memory to contain the edge connectivity array (num_edges_this_blk * num_edges_per_elem in length) and face connectivity arrays (num_edges_this_blk * num_faces_per_elem in length). If 0 or -1 was specified for num_edges_per_elem or num_faces_per_elem, then the respective values ofelem_edge_conn or elem_face_conn are ignored.

In case of an error, ex_get_conn returns a negative number; a warning will return a positive number. Zero is returned on success. Possible causes of errors include

- EXODUS file ID is invalid
- an block with the specified block type and ID is not stored in the file

ex_get_conn: C Interface

```
int ex_get_conn(exoid, blk_type, blk_id, node_conn);
```

`int exoid` **(R)**
EXODUS file ID returned from a previous call to ex_create or ex_open.

`int blk_type` **(R)**
One of EX_EDGE_BLOCK, EX_FACE_BLOCK, or EX_ELEM_BLOCK.

`int blk_id` **(R)**
The edge block ID whose connectivity is being retrieved.

`int node_conn[num_entries_this_blk,num_nodes_per_entry]` **(W)**
The returned connectivity array; a list of nodes (internal node IDs; see section 4.5) that define each entry in the block. The node index cycles faster than the edge, face, or element index. The type of entry (edge, face, or element) depends on the values of blk_type.

`int elem_edge_conn[num_elems_this_blk,num_edges_per_elem]` **(W)**
The returned edge connectivity array; a list of edges (internal edge IDs) that define each element in the block. This is ignored if blk_type is EX_EDGE_BLOCK or EX_FACE_BLOCK, or if num_edges_per_elem is 0 or -1. The edge index cycles faster than the element index.

`int elem_face_conn[num_elems_this_blk,num_faces_per_elem]` **(W)**

The returned face connectivity array; a list of faces (internal face IDs) that define each element in the block. This is ignored if `blk_type` is `EX_EDGE_BLOCK` or `EX_FACE_BLOCK`, or if `num_faces_per_elem` is 0 or -1. The face index cycles faster than the element index.

4.2.11 Write Edge, Face, or Element Block Attributes

The function ex_put_attr writes the attributes for a block of the type specified by blk_type. The argument blk_type must be one of EX_EDGE_BLOCK, EX_FACE_BLOCK, or EX_ELEM_BLOCK. Each entry in the block must have the same number of attributes, so there are (num_attr * num_edge_this_blk, num_attr * num_face_this_blk, or num_attr * num_elem_this_blk depending on the value of blk_type) attributes for each block. The function ex_put_block must be invoked with the same value of blk_type before this call is made.

Because the attributes are floating point values, the application code must declare the array passed to be the appropriate type ("float" or "double" in C) to match the compute word size passed in ex_create or ex_open.

In case of an error, ex_put_attr returns a negative number; a warning will return a positive number. Zero is returned on success. Possible causes of errors include

- data file not properly opened with a call to ex_create or ex_open
- data file opened for read only
- data file not initialized properly with call to ex_put_init (for EX_ELEM_BLOCK) or ex_put_init_ext (for all other values of blk_type)
- ex_put_block was not called previously for specified block type and block ID
- ex_put_block was called with 0 attributes specified

ex_put_attr: C Interface

```
int ex_put_attr(exoid, blk_type, blk_id, attrib);
```

`int exoid` **(R)**
EXODUS file ID returned from a previous call to ex_create or ex_open.

`int blk_type` **(R)**
One of EX_EDGE_BLOCK, EX_FACE_BLOCK, or EX_ELEM_BLOCK.

`int blk_id` **(R)**
The block ID whose atributes are being specified.

`const float/double attrib[num_entry_this_blk,num_attr]` **(R)**
The list of attributes for the block. The num_attr index cycles faster than the edge, face, or element index.

4.2.12 Read Edge, Face, or Element Block Attributes

The function ex_get_attr reads the attributes for an edge, face, or element block. The argument blk_type must be one of EX_EDGE_BLOCK, EX_FACE_BLOCK, or EX_ELEM_BLOCK, respectively. Memory must be allocated for (num_attr * num_edge_this_blk, num_attr * num_face_this_blk, or num_attr * num_elem_this_blk, respectively) attributes before this routine is called.

Because the attributes are floating point values, the application code must declare the array passed to be the appropriate type ("float" or "double" in C) to match the compute word size passed in ex_create or ex_open.

In case of an error, ex_get_attr returns a negative number; a warning will return a positive number. Zero is returned on success. Possible causes of errors include

- data file not properly opened with a call to ex_create or ex_open
- an invalid block ID
- a warning value is returned if no attributes are stored in the file

ex_get_attr: C Interface

```
int ex_get_attr(exoid, blk_type, blk_id, attrib);
```

`int exoid` **(R)**
EXODUS file ID returned from a previous call to ex_create or ex_open.

`int blk_type` **(R)**
One of EX_EDGE_BLOCK, EX_FACE_BLOCK, or EX_ELEM_BLOCK.

`int blk_id` **(R)**
The block ID whose attributes are being retrieved.

`float/double attrib[num_entries_this_blk,num_attr]` **(W)**
The returned list of attributes for the edge, face, or element block. The num_attr index cycles faster than the edge face or element index.

4.2.13 Write One Edge, Face, or Element Block Attribute

The function ex_put_one_attr writes a single attribute for an edge, face, or element block. The argument blk_type must be one of EX_EDGE_BLOCK, EX_FACE_BLOCK, or EX_ELEM_BLOCK, respectively. The function ex_put_block must be invoked before this call is made.

Because the attributes are floating point values, the application code must declare the array passed to be the appropriate type ("float" or "double" in C) to match the compute word size passed in ex_create or ex_open.

In case of an error, ex_put_one_attr returns a negative number; a warning will return a positive number. Zero is returned on success. Possible causes of errors include

- data file not properly opened with a call to ex_create or ex_open
- data file opened for read only
- data file not initialized properly with call to ex_put_init_ext
- ex_put_block was not called previously for specified block type and ID
- ex_put_block was called with 0 attributes specified
- the specified attribute index is larger than the number of attributes for this block type and ID.

ex_put_one_attr: C Interface

```
int ex_put_one_attr(exoid, blk_type, blk_id, attribute_index, attrib);
```

`int exoid` **(R)**
EXODUS file ID returned from a previous call to ex_create or ex_open.

`int blk_type` **(R)**
One of EX_EDGE_BLOCK, EX_FACE_BLOCK, or EX_ELEM_BLOCK.

`int blk_id` **(R)**
The block ID whose attribute is being specified.

`int attribute_index` **(R)**
The attribute of the given block whose values are specified by this call. This number should be in $\{1, \ldots,$ number of attributes in this block$\}$.

`const float/double attrib[num_entries_this_blk]` **(R)**
The list of attribute values for the specified block attribute.

4.2.14 Read One Edge, Face, or Element Block Attribute

The function ex_get_one_attr writes a single attribute for an edge, face, or element block. The argument blk_type must be one of EX_EDGE_BLOCK, EX_FACE_BLOCK, or EX_ELEM_BLOCK, respectively. Memory for (num_edge_this_blk, num_face_this_blk, or num_elem_this_blk depending on the value of blk_type) attribute values must be allocated before this call is made.

Because the attributes are floating point values, the application code must declare the array passed to be the appropriate type ("float" or "double" in C) to match the compute word size passed in ex_create or ex_open.

In case of an error, ex_get_one_attr returns a negative number; a warning will return a positive number. Zero is returned on success. Possible causes of errors include

- data file not properly opened with a call to ex_create or ex_open
- invalid block ID
- a warning value is returned if no attributes are stored in the file
- the attribute index is larger than the number of attributes for this block type and ID

ex_get_one_attr: C Interface

```
int ex_get_one_attr(exoid, blk_type, blk_id, attribute_index, attrib);
```

`int exoid` **(R)**
EXODUS file ID returned from a previous call to ex_create or ex_open.

`int blk_type` **(R)**
One of EX_EDGE_BLOCK, EX_FACE_BLOCK, or EX_ELEM_BLOCK.

`int blk_id` **(R)**
The block ID whose attribute is being retrieved.

`int attribute_index` **(R)**
The attribute of the given block whose values are specified by this call. This number should be in $\{1, \ldots,$ number of attributes in this block$\}$.

`float/double attrib[num_entries_this_blk]` **(W)**
The list of attribute values for the specified edge, face, or element block attribute.

4.2.15 Write Edge, Face, or Element Block Attribute Names

The function ex_put_attr_names writes the names of the attributes for a specified edge, face, or element block to the database. The argument blk_type must be one of EX_EDGE_BLOCK, EX_FACE_BLOCK, or EX_ELEM_BLOCK, respectively. The blocks must be defined via a call to ex_put_block or ex_put_concat_all_blocks before this call is made.

In case of an error, ex_put_attr_names returns a negative number; a warning will return a positive number. Zero is returned on success. Possible causes of errors include

- data file not properly opened with a call to ex_create or ex_open
- data file opened for read only
- data file not initialized properly with call to ex_put_init_ext
- ex_put_block was not called previously for specified block type and ID
- the specified block has zero attributes
- no block with the specified type and ID are present in the database

ex_put_attr_names: C Interface

```
int ex_put_attr_names(exoid, blk_type, block_id, attr_names);
```

`int exoid` **(R)**
: EXODUS file ID returned from a previous call to ex_create or ex_open.

`int blk_type` **(R)**
: One of EX_EDGE_BLOCK, EX_FACE_BLOCK, or EX_ELEM_BLOCK.

`int blk_id` **(R)**
: The block ID whose attribute names are being specified.

`const char** attr_names` **(R)**
: Array containing num_attr names (of length MAX_STR_LENGTH) of the attributes for the edge, face, or element block with ID block_id.

4.2.16 Read Edge, Face, or Element Block Attribute Names

The function ex_get_attr_names reads the names (MAX_STR_LENGTH-characters in length) of the attribute arrays from the database for the specified edge, face, or element block. The argument blk_type must be one of EX_EDGE_BLOCK, EX_FACE_BLOCK, or EX_ELEM_BLOCK, respectively. Memory must be allocated for the character strings before this function is invoked.

In case of an error, ex_get_attr_names returns a negative number; a warning will return a positive number. Zero is returned on success. Possible causes of errors include

- data file not properly opened with a call to ex_create or ex_open
- invalid block ID
- a warning value is returned if no attributes are stored in the file

ex_get_attr_names: C Interface

```
int ex_get_attr_names(exoid, blk_type, block_id, attr_names);
```

`int exoid` **(R)**
EXODUS file ID returned from a previous call to ex_create or ex_open.

`int blk_type` **(R)**
One of EX_EDGE_BLOCK, EX_FACE_BLOCK, or EX_ELEM_BLOCK.

`int blk_id` **(R)**
The block ID whose attribute names are being retrieved.

`char** attr_names` **(W)**
Returned pointer to a vector containing num_attr names of the element attributes for the edge, face, or element block with ID block_id.

4.2.17 Write Node, Edge, Face, Side, or Element Set Parameters

The function ex_put_set_param writes the node, edge, face, side, or element set ID and the number of nodes, edges, faces, sides, or elements which describe a single set, and the number of distribution factors on the set. The argument set_type must be one of EX_NODE_SET, EX_EDGE_SET, EX_FACE_SET, EX_SIDE_SET, or EX_ELEM_SET.

A node set is basically a subset of the nodes of a mesh that have some distribution factors defined over the subset.

An edge set is basically a subset of the edges of a mesh that have some distribution factors defined over the subset. Note that edge sets may refer to edges from multiple edge blocks since the edge numbers used are the local ordering across all edges in the file.

A face set is basically a subset of the faces of a mesh that have some distribution factors defined over the subset. Note that face sets may refer to faces from multiple face blocks since the face numbers used are the local ordering across all faces in the file.

A side set is basically a subset of the sides (edges or faces of elements) of a mesh that have some distribution factors defined over the subset. Note that side sets may refer to elements from multiple element blocks since the element numbers used are the local ordering across all elements in the file.

An element set is basically a subset of the elements of a mesh. Element sets are typically used by post-processing tools to denote regions of interest; pre-processing tools use element blocks rather than element sets to partition elements.

Perhaps it would be wise to change "distribution factor" to "boundary condition" since "distribution factor" has no meaning here? **Note**

In case of an error, ex_put_set_param returns a negative number; a warning will return a positive number. Zero is returned on success. Possible causes of errors include

- data file not properly opened with a call to ex_create or ex_open
- data file opened for read only
- data file not initialized properly with call to ex_put_init_ext
- the number of edge, face, or side sets specified in the call to ex_put_init_ext was zero or has been exceeded
- an edge, face, or side set with the specified ID has already been stored (note that an edge set, a face set, and a side set may all share the same ID, but two edge sets may not share the same ID)

ex_put_set_param: C Interface

```
int ex_put_set_param(exoid, set_type, set_id, num_entries_in_set, num_dist_fact_in_set);
```

`int exoid` **(R)**
EXODUS file ID returned from a previous call to `ex_create` or `ex_open`.

`int set_type` **(R)**
One of `EX_NODE_SET`, `EX_EDGE_SET`, `EX_FACE_SET`, `EX_SIDE_SET`, or `EX_ELEM_SET`.

`int set_id` **(R)**
The ID of the set being specified.

`int num_entries_in_set` **(R)**
The number of edges, faces, sides, or elements in the set.

`int num_dist_fact_in_set` **(R)**
The total number of distribution factors (`num_entries_in_set * num_dist_fact_per_entry`).

4.2.18 Read Node, Edge, Face, Side, or Element Set Parameters

The function ex_get_set_param reads the number of nodes, edges, faces, sides, or elements which describe a single set and the number of distribution factors on the set. The argument set_type must be one of EX_NODE_SET, EX_EDGE_SET, EX_FACE_SET, EX_SIDE_SET, or EX_ELEM_SET.

A node set is basically a subset of the nodes of a mesh that have some distribution factors defined over the subset.

An edge set is basically a subset of the edges of a mesh that have some distribution factors defined over the subset. Note that edge sets may refer to edges from multiple edge blocks since the edge numbers used are the local ordering across all edges in the file.

A face set is basically a subset of the faces of a mesh that have some distribution factors defined over the subset. Note that face sets may refer to faces from multiple face blocks since the face numbers used are the local ordering across all faces in the file.

A side set is basically a subset of the sides (edges or faces of elements) of a mesh that have some distribution factors defined over the subset. Note that side sets may refer to elements from multiple element blocks since the element numbers used are the local ordering across all elements in the file.

An element set is basically a subset of the elements of a mesh. They set may have some associated distribution factors defined. Note that element sets may refer to elements from multiple element blocks since the element numbers used are the local ordering across all elements in the file.

Perhaps it would be wise to change "distribution factor" to "boundary condition" since "distribution factor" has no meaning here? **Note**

In case of an error, ex_get_set_param returns a negative number; a warning will return a positive number. Zero is returned on success. Possible causes of errors include

- data file not properly opened with a call to ex_create or ex_open
- a warning value is returned if no sets are stored in the file
- invalid set type and/or ID

ex_get_set_param: C Interface

```
int ex_get_set_param(exoid, set_type, set_id, num_entries_in_set, num_dist_fact_in_set);
```

`int exoid` **(R)**
EXODUS file ID returned from a previous call to ex_create or ex_open.

`int set_type` **(R)**
: One of `EX_NODE_SET`, `EX_EDGE_SET`, `EX_FACE_SET`, `EX_SIDE_SET`, or `EX_ELEM_SET`.

`int set_id` **(R)**
: The ID of the set being specified.

`int* num_entries_in_set` **(W)**
: The number of edges, faces, sides, or elements in the set.

`int* num_dist_fact_in_set` **(W)**
: The total number of distribution factors (`num_entries_in_set * num_dist_fact_per_entry`).

4.2.19 Write a Node, Edge, Face, Side, or Element Set

The function ex_put_set writes an list of offsets and (for edge, face, or side sets only) extra data for a single set. The offsets are are references to entries in the list of all nodes, edges, faces, or elements contained in a file. The argument set_type must be one of EX_NODE_SET, EX_EDGE_SET, EX_FACE_SET, EX_SIDE_SET, or EX_ELEM_SET. The routine ex_put_set_param must be called before this function is invoked. When an edge or face set is written, both an offset and an orientation are provided for each entry in the set. When a side set is written, both an offset and a side (element-local edge or face index) are provided for each entry in the set.

In case of an error, ex_put_set returns a negative number; a warning will return a positive number. Zero is returned on success. Possible causes of errors include

- data file not properly opened with a call to ex_create or ex_open
- data file opened for read only
- data file not initialized properly with a call to ex_put_init_ext
- ex_put_set_param not called previously
- an inappropriate NULL pointer was passed

ex_put_set: C Interface

```
int ex_put_set(exoid, set_type, set_id, set_elem_list, set_extra_list);
```

`int exoid` **(R)**
: EXODUS file ID returned from a previous call to ex_create or ex_open.

`int set_type` **(R)**
: One of EX_NODE_SET, EX_EDGE_SET, EX_FACE_SET, EX_SIDE_SET, or EX_ELEM_SET.

`int set_id` **(R)**
: The ID of the set being specified.

`const int set_elem_list[num_entries_in_set]` **(R)**
: Array containing the nodes, edges, faces, or elements in the set.

`const int set_extra_list[num_entries_in_set]` **(R)**
: When set_type is EX_NODE_SET, this should be a NULL pointer. When set_type is EX_EDGE_SET or EX_FACE_SET, this array contains an orientation number (+1 or -1) for each entry in the edge or face set. When set_type is EX_SIDE_SET, this array contains a side index (an element-local edge or face number) for each entry in the side set.

4.2.20 Read a Node, Edge, Face, Side, or Element Set

The function ex_get_set reads an element list and (for edge, face, or side sets only) extra data for a single set. The argument set_type must be one of EX_NODE_SET, EX_EDGE_SET, EX_FACE_SET, EX_SIDE_SET, or EX_ELEM_SET. When an edge or face set is read, both an element list and an orientation are provided for each entry in the set. When a side set is read, both an element list and a side (edge or face) index are provided for each entry in the set. Memory must be allocated for the element list (num_node_in_set, num_edge_in_set, num_face_in_set, or num_side_in_set in length, respectively) and when set_type is EX_EDGE_SET, EX_FACE_SET, EX_SIDE_SET, or EX_ELEM_SET, for the extra data list (num_edge_in_set, num_face_in_set,or num_side_in_set in length, respectively) before this function is invoked.

In case of an error, ex_get_set returns a negative number; a warning will return a positive number. Zero is returned on success. Possible causes of errors include

- data file not properly opened with a call to ex_create or ex_open
- a warning value is returned if no sets are stored in the file
- an invalid set ID is given

ex_get_set: C Interface

```
int ex_get_set(exoid, set_type, set_id, set_elem_list, set_extra_list);
```

`int exoid` **(R)**
EXODUS file ID returned from a previous call to ex_create or ex_open.

`int set_type` **(R)**
One of EX_NODE_SET, EX_EDGE_SET, EX_FACE_SET, EX_SIDE_SET, or EX_ELEM_SET.

`int set_id` **(R)**
The ID of the set to be read.

`int set_elem_list[num_entries_in_set]` **(W)**
Array where the nodes, edges, faces, or elements in the set will be stored.

`int set_extra_list[num_entries_in_set]` **(W)**
When set_type is EX_NODE_SET, this argument is ignored and a NULL pointer should be passed. When set_type is EX_EDGE_SET or EX_FACE_SET, an orientation number (+1 or -1) will be stored for each set entry. When set_type is EX_SIDE_SET, a side index (an element-local edge or face number) will be stored for each set entry.

See the documentation of ex_get_set_param for an example of ex_get_set.

4.2.21 Write Node, Edge, Face, Side, or Element Set Distribution Factors

The function ex_put_set_dist_fact writes distribution factors for a single node, edge, face, side, or element set. The argument set_type must be one of EX_NODE_SET, EX_EDGE_SET, EX_FACE_SET, EX_SIDE_SET, or EX_ELEM_SET. The routine ex_put_set_param must be called for the set before this function is invoked.

Because the attributes are floating point values, the application code must declare the array passed to be the appropriate type ("float" or "double" in C) to match the compute word size passed in ex_create or ex_open.

In case of an error, ex_put_set_dist_fact returns a negative number; a warning will return a positive number. Zero is returned on success. Possible causes of errors include

- data file not properly opened with a call to ex_create or ex_open
- data file opened for read only
- data file not initialized properly with a call to ex_put_init_ext
- ex_put_set_param not called previously

ex_put_set_dist_fact: C Interface

```
int ex_put_set_dist_fact(exoid, set_type, set_id, set_dist_fact);
```

`int exoid` **(R)**
: EXODUS file ID returned from a previous call to ex_create or ex_open.

`int set_type` **(R)**
: One of EX_NODE_SET, EX_EDGE_SET, EX_FACE_SET, EX_SIDE_SET, or EX_ELEM_SET.

`int set_id` **(R)**
: The ID of the set being specified.

`const float/double* set_dist_fact` **(R)**
: Array containing the distribution factors for the set.

4.2.22 Read Node, Edge, Face, Side, or Element Set Distribution Factors

The function ex_get_set_dist_fact writes distribution factors for a single node, edge, face, side, or element set. The argument set_type must be one of EX_NODE_SET, EX_EDGE_SET, EX_FACE_SET, EX_SIDE_SET, or EX_ELEM_SET. Memory must be allocated for the element list (num_dist_fact_in_set in length) before this function is invoked. The routine ex_get_set_param must be called before this function is invoked.

Because the attributes are floating point values, the application code must declare the array passed to be the appropriate type ("float" or "double" in C) to match the compute word size passed in ex_create or ex_open.

In case of an error, ex_get_set_dist_fact returns a negative number; a warning will return a positive number. Zero is returned on success. Possible causes of errors include

- data file not properly opened with a call to ex_create or ex_open
- a warning value is returned if no sets are stored in the file
- an invalid set type or ID is given

ex_get_set_dist_fact: C Interface

```
int ex_get_set_dist_fact(exoid, set_type, set_id, set_dist_fact);
```

`int exoid` **(R)**
: EXODUS file ID returned from a previous call to ex_create or ex_open.

`int set_type` **(R)**
: One of EX_NODE_SET, EX_EDGE_SET, EX_FACE_SET, EX_SIDE_SET, or EX_ELEM_SET.

`int set_id` **(R)**
: The ID of the set being specified.

`float/double* set_dist_fact` **(W)**
: Array to hold the distribution factors for the set.

4.2.23 Get Node, Edge, Face, Side, or Element Set Node List Length

The `ex_get_set_node_list_len` function is not currently necessary. Although it may be implemented in the future, it need not be implemented for initial edge and face support.

4.2.24 Read Node, Edge, Face, Side, or Element Set Node List

The `ex_get_set_node_list` function is not currently necessary. Although it may be implemented in the future, it need not be implemented for initial edge and face support.

4.2.25 Write Concatenated Node, Edge, Face, Side, or Element Sets

For a given type of set (EX_NODE_SET, EX_EDGE_SET, EX_FACE_SET, EX_SIDE_SET, or EX_ELEM_SET), the function ex_put_concat_sets writes the the set IDs, the array of entry counts per set, the array of distribution factor counts per set, the offsets of each set into the concatenated node, edge, face, or element list, the offsets of each set into the concatenated orientation (for edge and face sets) or side (for side sets) list, the offsets of each set's distribution factors into the concatenated distribution factor list, the concatenated list, and the concatenated distribution factor list. "Concatenated set" refers to all of the arrays (listed above) that are needed to define all of the sets of a given type. Writing concatenated sets is more efficient than writing individual sets.

Because the attributes are floating point values, the application code must declare the array passed to be the appropriate type ("float" or "double" in C) to match the compute word size passed in ex_create or ex_open.

It is possible to use this call to only *d*efine the sets on the database and to *w*rite the sets using other API functions. This usage is also more efficient than defining individual sets, but sometimes easier than defining and writing all set data at one time. To only define the sets on the database, pass a NULL for the sets_entry_index, sets_dist_index, sets_entry_list, sets_extra_list, and sets_dist_fact arguments.

When using this call to write node sets, pass a NULL for the sets_extra_list.

Rather than requiring a long list of arguments, ex_put_concat_sets takes a C structure named ex_set_specs containing all of the information about the various sets. The structure is defined as

ex_set_specs: C Structure

```
struct ex_set_specs {
```

int* sets_ids
: An array containing the ID of each set.

int* num_entries_per_set
: An array containing the number of entries (nodes, edges, faces, sides, or elements) in each set.

int* num_dist_per_set
: An array containing the number of distribution factors in each set.

int* sets_entry_index
: An array containing the offset (into sets_entry_list and sets_extra_list if appropriate) of the first entry in each set. Pass NULL if only *defining* sets with this call.

int* sets_dist_index
: An array containing the offset (into sets_dist_fact) of the first distribution factor in each edge set. Pass NULL if only *defining* sets with this call.

int* sets_entry_list
: The list of all nodes, edges, faces, or elements in all sets concatenated into

a single array. Each set's entries are stored contiguously but sets may be ordered arbitrarily. Pass NULL if only *defining* sets with this call.

`int* sets_extra_list`
: The list of all orientations (values of +1 or -1 for edge or face sets), or sides (for side sets) in all sets concatenated into a single array. Each set's entries are stored contiguously but sets may be ordered arbitrarily. Pass NULL if `set_type` is `EX_NODE_SET` or `EX_ELEM_SET` or if only *defining* sets with this call.

`float/double* sets_dist_fact`
: The list of all distribution factors in all sets concatenated into a single array. Each set's distribution factors are stored contiguously but the sets may be ordered arbitrarily. Pass NULL if only *defining* sets with this call.

}

In case of an error, `ex_put_concat_sets` returns a negative number; a warning will return a positive number. Zero is returned on success. Possible causes of errors include

- data file not properly opened with a call to `ex_create` or `ex_open`
- data file opened for read only
- data file not initialized properly with a call to `ex_put_init_ext`
- the number of sets specified in a call to `ex_put_init_ext` was zero or has been exceeded
- a set with the same type or ID has already been stored.
- an invalid set type was specified

ex_put_concat_sets: C Interface

```
int ex_put_concat_sets(exoid, set_type, set_data);
```

`int exoid` **(R)**
: EXODUS file ID returned from a previous call to `ex_create` or `ex_open`.

`int set_type` **(R)**
: One of `EX_NODE_SET`, `EX_EDGE_SET`, `EX_FACE_SET`, `EX_SIDE_SET`, or `EX_ELEM_SET`.

`const ex_set_specs* set_data` **(R)**
: A pointer to a valid `ex_set_specs` structure containing the data to be written.

4.2.26 Read Concatenated Node, Edge, Face, Side, or Element Sets

The function ex_get_concat_sets reads the the set IDs, the array of node, edge, face, side, or element counts per set, the array of distribution factor counts per set, the offsets of each set into the concatenated node, edge, face, or element list, the offsets of each set into the concatenated orientation (for edge and face sets) or side (for side sets) list, the offsets of each set's distribution factors into the concatenated distribution factor list, the concatenated list, and the concatenated distribution factor list. "Concatenated set" refers to all of the arrays (listed above) that are needed to define all of the sets of a given type. Memory must be allocated for each of these arrays before ex_get_concat_sets is invoked. Use ex_get_init_ext and ex_inquire to determine the required lengths of the arrays.

Because the attributes are floating point values, the application code must declare the array passed to be the appropriate type ("float" or "double" in C) to match the compute word size passed in ex_create or ex_open.

Rather than requiring a long list of arguments, ex_get_concat_sets takes a C structure named ex_set_specs containing all of the information about the various sets. The structure is defined in §4.2.25.

In case of an error, ex_get_concat_edge_sets returns a negative number; a warning will return a positive number. Zero is returned on success. Possible causes of errors include

- data file not properly opened with a call to ex_create or ex_open
- a warning value is returned if no edge sets are stored in the file

ex_get_concat_edge_sets: C Interface

```
int ex_get_concat_edge_sets(exoid, set_type, set_data);
```

`int exoid` **(R)**
: EXODUS file ID returned from a previous call to ex_create or ex_open.

`int set_type` **(R)**
: One of EX_NODE_SET, EX_EDGE_SET, EX_FACE_SET, EX_SIDE_SET, or EX_ELEM_SET.

`ex_set_specs* set_data` **(RW)**
: A pointer to an ex_set_specs structure containing pointers to arrays which will be filled with the requested sets.

4.2.27 Modified! Write Object Names

Note

Object now encompasses a larger array of block, set, and map types. Therefore, the `obj_type` argument must now also accept the following additional values:

- `EX_EDGE_BLOCK` To designate an edge block.
- `EX_FACE_BLOCK` To designate an face block.
- `EX_EDGE_SET` To designate an edge set.
- `EX_FACE_SET` To designate an face set.
- `EX_ELEM_SET` To designate an element set.
- `EX_EDGE_MAP` To designate an edge map.
- `EX_FACE_MAP` To designate an face map.

4.2.28 Modified! Read Object Names

Note

Object now encompasses a larger array of block, set, and map types. Therefore, the `obj_type` argument may now take on the following additional values:

- `EX_EDGE_BLOCK` To designate an edge block.
- `EX_FACE_BLOCK` To designate an face block.
- `EX_EDGE_SET` To designate an edge set.
- `EX_FACE_SET` To designate an face set.
- `EX_ELEM_SET` To designate an element set.
- `EX_EDGE_MAP` To designate an edge map.
- `EX_FACE_MAP` To designate an face map.

4.2.29 Modified! Write Individual Object Name

Note

Object now encompasses a larger array of block, set, and map types. Therefore, the `obj_type` argument must now also accept the following additional values:

- `EX_EDGE_BLOCK` To designate an edge block.
- `EX_FACE_BLOCK` To designate an face block.
- `EX_EDGE_SET` To designate an edge set.
- `EX_FACE_SET` To designate an face set.
- `EX_ELEM_SET` To designate an element set.
- `EX_EDGE_MAP` To designate an edge map.
- `EX_FACE_MAP` To designate an face map.

4.2.30 Modified! Read Individual Object Name

Note

Object now encompasses a larger array of block, set, and map types. Therefore, the `obj_type` argument may now take on the following additional values:

- `EX_EDGE_BLOCK` To designate an edge block.
- `EX_FACE_BLOCK` To designate an face block.
- `EX_EDGE_SET` To designate an edge set.
- `EX_FACE_SET` To designate an face set.
- `EX_ELEM_SET` To designate an element set.
- `EX_EDGE_MAP` To designate an edge map.
- `EX_FACE_MAP` To designate an face map.

4.2.31 Modified! Write Property Arrays Names

Note

Object now encompasses a larger array of block, set, and map types. Therefore, the `obj_type` argument must now also accept the following additional values:

- `EX_EDGE_BLOCK` To designate an edge block.
- `EX_FACE_BLOCK` To designate an face block.
- `EX_EDGE_SET` To designate an edge set.
- `EX_FACE_SET` To designate an face set.
- `EX_ELEM_SET` To designate an element set.
- `EX_EDGE_MAP` To designate an edge map.
- `EX_FACE_MAP` To designate an face map.

4.2.32 Modified! Read Property Arrays Names

Note

Object now encompasses a larger array of block, set, and map types. Therefore, the `obj_type` argument may now take on the following additional values:

- `EX_EDGE_BLOCK` To designate an edge block.
- `EX_FACE_BLOCK` To designate an face block.
- `EX_EDGE_SET` To designate an edge set.
- `EX_FACE_SET` To designate an face set.
- `EX_ELEM_SET` To designate an element set.
- `EX_EDGE_MAP` To designate an edge map.
- `EX_FACE_MAP` To designate an face map.

4.2.33 Modified! Write Object Property

Note

Object now encompasses a larger array of block, set, and map types. Therefore, the `obj_type` argument must now also accept the following additional values:

- `EX_EDGE_BLOCK` To designate an edge block.
- `EX_FACE_BLOCK` To designate an face block.
- `EX_EDGE_SET` To designate an edge set.
- `EX_FACE_SET` To designate an face set.
- `EX_ELEM_SET` To designate an element set.
- `EX_EDGE_MAP` To designate an edge map.
- `EX_FACE_MAP` To designate an face map.

4.2.34 Modified! Read Object Property

Object now encompasses a larger array of block, set, and map types. Therefore, the `obj_type` argument may now take on the following additional values:

Note

- `EX_EDGE_BLOCK` To designate an edge block.
- `EX_FACE_BLOCK` To designate an face block.
- `EX_EDGE_SET` To designate an edge set.
- `EX_FACE_SET` To designate an face set.
- `EX_ELEM_SET` To designate an element set.
- `EX_EDGE_MAP` To designate an edge map.
- `EX_FACE_MAP` To designate an face map.

4.2.35 Modified! Write Object Property Array

Note

Object now encompasses a larger array of block, set, and map types. Therefore, the `obj_type` argument must now also accept the following additional values:

- `EX_EDGE_BLOCK` To designate an edge block.
- `EX_FACE_BLOCK` To designate an face block.
- `EX_EDGE_SET` To designate an edge set.
- `EX_FACE_SET` To designate an face set.
- `EX_ELEM_SET` To designate an element set.
- `EX_EDGE_MAP` To designate an edge map.
- `EX_FACE_MAP` To designate an face map.

4.2.36 Modified! Read Object Property Array

Object now encompasses a larger array of block, set, and map types. Therefore, the `obj_type` argument may now take on the following additional values:

Note

- `EX_EDGE_BLOCK` To designate an edge block.
- `EX_FACE_BLOCK` To designate an face block.
- `EX_EDGE_SET` To designate an edge set.
- `EX_FACE_SET` To designate an face set.
- `EX_ELEM_SET` To designate an element set.
- `EX_EDGE_MAP` To designate an edge map.
- `EX_FACE_MAP` To designate an face map.

4.3 Results Data

Results data is that part of the finite element data allowed to vary over time. Edges and faces can have an arbitrary number of variables defined over edge blocks and face blocks. Edge, face, and element sets can also have variables. The "results variables parameters" routines below allow the developer to specify which variables are defined over which objects (file (global), nodes, edges, faces, elements, node sets, edge sets, face sets, side sets, and element sets). Truth tables are maintained for each object type (except nodal and global variables) so that variable values need not be stored for every block or set.

4.3.1 Modified! Write Results Variables Parameters

Note

Results variables can now be defined over edge blocks and edge sets, face blocks and face sets. This means that `ex_put_var_param` must accept the following additional values for `var_type`:

"ℓ" (or "L") For edge variables.
"f" (or "F") For face variables.
"d" (or "D") For eDge set variables.
"a" (or "A") For fAce set variables.
"t" (or "T") For elemenT set variables.

4.3.2 Modified! Read Results Variables Parameters

Note

Results variables can now be defined over edge blocks and edge sets, face blocks and face sets. This means that `ex_get_var_param` must accept the following additional values for `var_type`:

"ℓ" (or "L")	For edge variables.
"f" (or "F")	For face variables.
"d" (or "D")	For eDge set variables.
"a" (or "A")	For fAce set variables.
"t" (or "T")	For elemenT set variables.

4.3.3 Modified! Write All Results Variables Parameters

Note

Results variables can now be defined over edge blocks and edge sets, face blocks and face sets, and element sets in addition to the previous possibilities. This means that the `elem_var_tab` and `sset_var_tab` truth tables must be enlarged or that new truth tables for variables defined over these new object types must be defined. Because edges and faces are turned into first-class objects by these API extensions, we choose to define new tables: `edge_var_tab(num_edge_blk,num_edge_var)`, `face_var_tab(num_face_blk,num_face_var)`, `eset_var_tab(num_edge_set,num_es_var)`, `fset_var_tab(num_face_set,num_fs_var)`, and `elset_var_tab(num_elace_set,num_els_var)`. Rather than change the existing EXODUS API, we add the following call.

The function `ex_put_all_var_param_ext` defines in one call the number of global, nodal, nodeset, edge set, face set, sideset, element set, edge, face, and element variables that will be written to the database. Using this function is more efficient than calling `ex_put_var_param` for global, nodal, nodeset, edge set, face set, sideset, element set, edge, face, and element variables followed by a call to put the node set, edge set, face set, sideset, element set, edge, face, and element truth tables. See Appendix A of the EXODUS manual for a more in-depth description of EXODUS efficiency concerns.

Rather than passing a long list of arguments, a C structure is used to store the results variables parameters:

ex_var_params: C Structure

```
struct ex_var_params {
```

`int num_glob`
: The number of global variables that will be written to the database.

`int num_node`
: The number of nodal variables that will be written to the database.

`int num_edge`
: The number of edge block variables that will be written to the database.

`int* edge_var_tab`
: A 2-dimensional array of size `num_edge_blk`×`num_edge_var` (with `num_edge_var` index cycling faster) containing the edge variable truth table.

`int num_face`
: The number of face block variables that will be written to the database.

`int* face_var_tab`
: A 2-dimensional array of size `num_face_blk`×`num_face_var` (with `num_face_var` index cycling faster) containing the face variable truth table.

`int num_elem`
: The number of element block variables that will be written to the database.

`int* elem_var_tab`
: A 2-dimensional array of size `num_elem_blk`×`num_elem_var` (with `num_elem_var` index cycling faster) containing the element variable truth table.

`int num_nset`
: The number of node set variables that will be written to the database.

`int* nset_var_tab`
: A 2-dimensional array of size `num_node_set`×`num_nset_var` (with `num_nset_var` index cycling faster) containing the node set variable truth table.

`int num_eset`
: The number of edge set variables that will be written to the database.

`int* eset_var_tab`
: A 2-dimensional array of size `num_edge_set`×`num_eset_var` (with `num_eset_var` index cycling faster) containing the edge set variable truth table.

`int num_fset`
: The number of face set variables that will be written to the database.

`int* fset_var_tab`
: A 2-dimensional array of size `num_face_set`×`num_fset_var` (with `num_fset_var` index cycling faster) containing the face set variable truth table.

`int num_sset`
: The number of side set variables that will be written to the database.

`int* sset_var_tab`
: A 2-dimensional array of size `num_side_set`×`num_sset_var` (with `num_sset_var` index cycling faster) containing the side set variable truth table.

`int num_elset`
: The number of element set variables that will be written to the database.

`int* elset_var_tab`
: A 2-dimensional array of size `num_elem_set`×`num_elset_var` (with `num_elset_var` index cycling faster) containing the element set variable truth table.

}

In case of an error, `ex_put_all_var_param_ext` returns a negative number; a warning will return a positive number. Zero is returned on success. Possible causes of errors include:

- data file not properly opened with a call to `ex_create` or `ex_open`
- data file opened for read only
- data file not properly initialized with a call to `ex_put_init_ext`
- this routine has already been called (redefining the number of variables is not allowed)
- a warning is returned if the number of variables is specified as zero.

ex_put_all_var_param_ext: C Interface

```
int ex_put_all_var_param_ext(exoid, ex_vparam);
```

`int exoid` **(R)**
: EXODUS file ID returned from a previous call to `ex_create` or `ex_open`.

`const ex_var_params ex_vparam` **(R)**

A structure containing results variables parameter values. See the description of `ex_var_params` above for details.

4.3.4 Modified! Write Results Variables Names

Note

Results variables can now be defined over edge blocks and edge sets, face blocks, face sets, and element sets. This means that `ex_put_var_names` must accept the following additional values for `var_type`:

"ℓ" (or "L")	For edge variables.
"f" (or "F")	For face variables.
"d" (or "D")	For eDge set variables.
"a" (or "A")	For fAce set variables.
"t" (or "T")	For elemenT set variables.

4.3.5 Modified! Read Results Variables Names

Note

Results variables can now be defined over edge blocks and edge sets, face blocks, face sets, and element sets. This means that `ex_get_var_names` must accept the following additional values for `var_type`:

"ℓ" (or "L") For edge variables.

"f" (or "F") For face variables.

"d" (or "D") For eDge set variables.

"a" (or "A") For fAce set variables.

"t" (or "T") For elemenT set variables.

4.3.6 Modified! Write Individual Results Variables Names

Note

Results variables can now be defined over edge blocks and edge sets, face blocks, face sets, and element sets. This means that `ex_put_var_name` must accept the following additional values for `var_type`:

"ℓ" (or "L")	For edge variables.
"f" (or "F")	For face variables.
"d" (or "D")	For eDge set variables.
"a" (or "A")	For fAce set variables.
"t" (or "T")	For elemenT set variables.

4.3.7 Modified! Read Individual Results Variables Names

Note

Results variables can now be defined over edge blocks and edge sets, face blocks, face sets, and element sets. This means that `ex_get_var_name` must accept the following additional values for `var_type`:

"ℓ" (or "L") For edge variables.
"f" (or "F") For face variables.
"d" (or "D") For eDge set variables.
"a" (or "A") For fAce set variables.
"t" (or "T") For elemenT set variables.

4.3.8 Modified! Write Object Variable Truth Table

Note

Results variables can now be defined over edge blocks and edge sets, face blocks, face sets, and element sets. This means that `ex_put_var_tab` must accept the following additional values for `var_type`:

"ℓ" (or "L")	For edge variables.
"f" (or "F")	For face variables.
"d" (or "D")	For eDge set variables.
"a" (or "A")	For fAce set variables.
"t" (or "T")	For elemenT set variables.

4.3.9 Modified! Read Object Variable Truth Table

Note

Results variables can now be defined over edge blocks and edge sets, face blocks, face sets, and element sets. This means that `ex_get_var_tab` must accept the following additional values for `var_type`:

"ℓ" (or "L") For edge variables.

"f" (or "F") For face variables.

"d" (or "D") For eDge set variables.

"a" (or "A") For fAce set variables.

"t" (or "T") For elemenT set variables.

4.3.10 Write Edge, Face, or Element Variable Values on Blocks or Sets at a Time Step

The function ex_put_var writes the values of a single edge, face, or element variable for one block or set at one time step. The var_type argument must be one of EX_EDGE_BLOCK, EX_FACE_BLOCK, EX_ELEM_BLOCK, EX_NODE_SET, EX_EDGE_SET, EX_FACE_SET, EX_SIDE_SET, EX_ELEM_SET, or EX_GLOBAL. It is recommended, but not required, to write the corresponding edge, face, or element variable truth table (with ex_put_var_tab) before this function is invoked for better efficiency.

Normally, ex_put_var writes values for a single variable. However, when var_type is EX_GLOBAL, you may write multiple variable values at once. To write a single global variable value, pass var_index set to the variable number and num_entries_this_object set to 1. To write all global variable values for a given time step, pass var_index set to 1 and num_entries_this_object set to the number of global variables.

Because variables are floating point values, the application code must declare the array passed to be the appropriate type ("float" or "double") to match the compute word size passed in ex_create or ex_open.

In case of an error, ex_put_var returns a negative number; a warning will return a positive number. Zero is returned on success. Possible causes of errors include:

- data file not properly opened with a call to ex_create or ex_open
- data file opened for read only
- data file not properly initialized with a call to ex_put_init_ext
- invalid variable type or ID
- ex_put_block/ex_put_set not called previously to specify parameters for this block/set
- ex_put_var_param_ext not called previously specifying the number of edge or face variables or ex_put_var_param not called previously specifying the number of variables
- a variable truth table was stored in the file but contains a zero (indicating no valid variable) for the specified object (block or set) and variable.

ex_put_var: C Interface

```
int ex_put_var(exoid, time_step, var_type, var_index,
obj_id, num_entries_this_var, var_vals);
```

`int exoid` **(R)**
: EXODUS file ID returned from a previous call to ex_create or ex_open.

`int time_step` **(R)**
: The time step number as described under ex_put_time. This is essentially a counter that is incremented only when results variables are output. The first time step is 1.

`int var_type` **(R)**

One of `EX_EDGE_BLOCK`, `EX_FACE_BLOCK`, `EX_ELEM_BLOCK`, `EX_NODE_SET`, `EX_EDGE_SET`, `EX_FACE_SET`, `EX_SIDE_SET`, `EX_ELEM_SET`, or `EX_GLOBAL`.

`int var_index` **(R)**

The index of the variable in the list of all variables of type `var_type`. The first variable has an index of 1.

`int obj_id` **(R)**

The block or set ID. This argument is ignored when `var_type` is `EX_GLOBAL`.

`int num_entries_this_var` **(R)**

The number of entries in the given block or set.

`const float/double* var_vals` **(R)**

Array of `num_entries_this_var` values of the `var_index`-th variable for the block or set with an ID of `var_id` at the `time_step`-th time step.

4.3.11 Read Edge, Face, or Element Variable Values Defined On Blocks or Sets at a Time Step

The function ex_get_var reads the values of a single edge, face, or element variable for one block or set at one time step. The var_type argument must be one of EX_EDGE_BLOCK, EX_FACE_BLOCK, EX_ELEM_BLOCK, EX_NODE_SET, EX_EDGE_SET, EX_FACE_SET, EX_SIDE_SET, or EX_ELEM_SET. Memory must be allocated for the variable values array before this function is invoked.

Because variables are floating point values, the application code must declare the array passed to be the appropriate type ("float" or "double") to match the compute word size passed in ex_create or ex_open.

In case of an error, ex_get_var returns a negative number; a warning will return a positive number. Zero is returned on success. Possible causes of errors include:

- data file not properly opened with a call to ex_create or ex_open
- variable does not exist for the desired block or set
- invalid variable type or ID.

ex_get_var: C Interface

```
int ex_get_var(exoid, time_step, var_type, var_index,
var_id, num_entries_this_var, var_vals);
```

`int exoid` **(R)**
: EXODUS file ID returned from a previous call to ex_create or ex_open.

`int time_step` **(R)**
: The time step number as described under ex_put_time. This is essentially a counter that is incremented only when results variables are output. The first time step is 1.

`int var_type` **(R)**
: One of EX_EDGE_BLOCK, EX_FACE_BLOCK, EX_ELEM_BLOCK, EX_NODE_SET, EX_EDGE_SET, EX_FACE_SET, EX_SIDE_SET, or EX_ELEM_SET.

`int var_index` **(R)**
: The index of the variable in the list of all variables of type var_type. The first variable has an index of 1.

`int var_id` **(R)**
: The block or set ID.

`int num_entries_this_var` **(R)**
: The number of entries in the given block or set.

`float/double* var_vals` **(W)**
: Array of num_entries_this_var values of the var_index-th variable for the block or set with an ID of var_id at the time_step-th time step.

4.3.12 Read Edge, Face, or Element Variable Values Defined On Blocks or Sets Through Time

The function ex_get_var_time reads the values of an edge, face, or element variable for a single edge, face, or element/side through the specified time step range. Memory must be allocated for the variable values array before this function is invoked.

Because variables are floating point values, the application code must declare the array passed to be the appropriate type ("float" or "double") to match the compute word size passed in ex_create or ex_open.

In case of an error, ex_get_var returns a negative number; a warning will return a positive number. Zero is returned on success. Possible causes of errors include:

- data file not properly opened with a call to ex_create or ex_open
- data file not initialized properly with a call to ex_put_init_ext
- variable does not exist for the desired edge, face, or element/side or results haven't been written
- invalid variable type or ID.

ex_get_var_time: C Interface

```
int ex_get_var_time(exoid, var_type, var_index, entry_number,
beg_time_step, end_time_step, var_vals);
```

`int exoid` **(R)**
EXODUS file ID returned from a previous call to ex_create or ex_open.

`int var_type` **(R)**
One of EX_EDGE_BLOCK, EX_FACE_BLOCK, EX_ELEM_BLOCK, EX_NODE_SET, EX_EDGE_SET, EX_FACE_SET, EX_SIDE_SET, or EX_ELEM_SET.

`int var_index` **(R)**
The index of the desired variable in the list of all variables of type var_type. The first variable has an index of 1.

`int entry_number` **(R)**
The internal ID (see the EXODUS manual Section 4.5) of the desired edge, face, or element. The first entry is 1.

`int beg_time_step` **(R)**
The beginning time step for which the variable value is desired. This is not a time value but rather a time step number, as described under ex_put_time. The first time step is 1.

`int end_time_step` **(R)**
The last time step for which the variable value is desired. If negative, the last time step in the database will be used. This is not a time value but rather a time step number, as described under ex_put_time. The first time step is 1.

`float/double* var_vals` **(W)**

Array of (`end_time_step` - `beg_time_step` + 1) values of the `var_index`-th variable in the block or set specified by `var_type` on the edge, face, or element with an internal ID of `entry_number`.

This page intentionally left blank

5 Conclusion

This document provides a specification for the storage of edge and face element data in the EXODUS file format. No changes to the EXODUS API are required, but a convention for specifying edges of three-dimensional finite elements in side sets is required. Such a convention is put forth in this document. Some minor alterations to VTK's EXODUS reader are required in order to display the edge circulations and face fluxes saved to disk. For the interpolation of the resulting vector field to points inside the mesh, further work is required.

References

[1] P. B. Bočev, J. J . Hu, A. C. Robinson, and R. S. Tuminaro. Towards robust 3d z-pinch simulations: discretization and fast solvers for magnetic diffusion in heterogeneous conductors. *Electronic Transactions on Numerical Analysis*, 15:186–210, 2003.

[2] R. D. Cook, D. S. Malkus, and M. E. Plesha. *Concepts and Applications of Finite Element Analysis*. John-Wiley, third edition, 1989.

[3] Y. Kuznetsov, K. Lipnikov, and M. Shashkov. The mimetic finite difference method on polygonal meshes for diffusion-type problems. *Computational Geosciences*, 8(4):301 – 324, Dec 2004.

[4] G. D. Sjaardema, L. A. Schoof, and V. R. Yarberry. EXODUS II: A finite element data model. Technical Report SAND92-2137, March 21 2006.

[5] Barna Szabo and Ivo Babuska. *Finite Element Analysis*. John Wiley & Sons, 1991.

[6] D. C. Thompson, R. Crawford, R. Khardekar, and P. P. Pébay. Visualization of higher order finite elements. Sandia Report SAND2004-1617, Sandia National Laboratories, April 2004.

DISTRIBUTION:

1 MS 0378
Allen Robinson, 1431

1 MS 0378
Richard Drake, 1431

1 MS 0380
Garth Reese, 1542

1 MS 0382
Greg Sjaardema, 1543

1 MS 0382
Kevin D. Copps, 1543

1 MS 9051
Philippe P. Pébay, 8351

1 MS 9012
David C. Thompson, 8963

1 MS 9012
Jeff N. Jortner, 8963

1 MS 9051
Dawn Manley, 8351

1 MS 9012
Jerry A. Friesen, 8963

1 MS 0822
Brian N. Wylie, 1424

1 MS 0822
David H. Rogers, 1424

1 MS 0822
Rena A. Haynes, 1424

1 MS 0376
Darryl J. Melander, 1421

2 MS 9018
Central Technical Files, 8944

2 MS 0899
Technical Library, 4536