BLUEBIRD DEVELOPER MANUAL

Author: James Craig

October, 2002

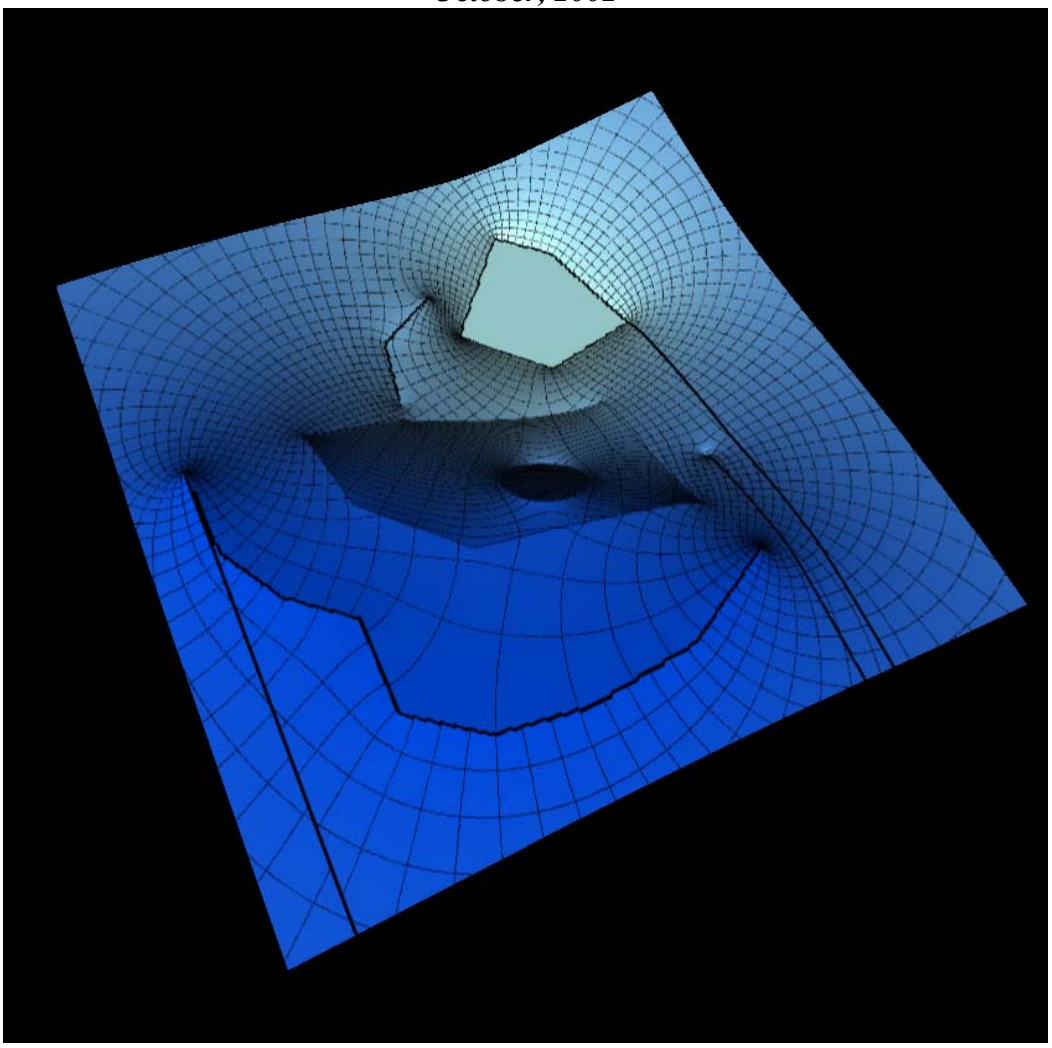


Table of Contents

1.0 Introduction

Bluebird is an object-oriented library of object classes, data, and functionality for 2-Dimensional modeling of steady-state saturated groundwater flow. It was designed as a development tool for improving the analytic element method and building complicated models in an automated fashion. This manual describes the scope of the library, some details about the analytic element method, and the structure of object orientation so that multiple developers may use Bluebird for their own research and modeling needs.

1.1 The 2-D Analytic Element method

The analytic element method is an alternative to finite element or finite difference solution of saturated groundwater flow problems. The method is based upon the superposition of analytic functions, called "analytic elements", each corresponding to a hydrogeologic feature (or portion of a feature) in the model, such as a river, inhomogeneity in conductivity, or pumping well. The superposed solution of these functions provides the pressure head and discharge at every point in the domain as a smooth, continuous function that is discontinuous only at element boundaries.

Instead of formulating the analytic solutions for groundwater flow in terms of head as a function of position: $\phi(x, y)$, the analytic element method relies upon superposition of functions in terms of discharge potential, Φ. In two dimensions, this discharge potential is formulated in terms of head so that the following relationship is always valid for irrotational flow:

$$Q_x = -\frac{\partial \Phi}{\partial x} \qquad Q_y = -\frac{\partial \Phi}{\partial y}$$

Where Qx and Qy are the components of the discharge vector in the x and y directions, and represent flux of water through a unit-thickness strip that extends vertically across the entire thickness of the aquifer. This formulation is equally valid for both confined and unconfined flow. In addition to formulation in terms of discharge potential, the 2-dimensional analytic element method relies upon the Dupuit-Forcheimer assumption, which states that there is no resistance to flow in the vertical direction (i.e. $k_z=\infty$). This allows for development of complex-valued potential functions that represent a wide variety of boundary condition types and geometries. Each hydrogeologic feature in the model is associated with such a function.

The analytic element method differs from classical analytic solutions in that each elementary solution (element) has a number of degrees of freedom (coefficients). Regardless of the choice of element coefficients, away from element borders the governing equations of mass continuity and Darcy's law are met *exactly*. These coefficients are chosen in such a way that internal boundary conditions (e.g. specified head) are met along the borders of elements, thus providing a complete solution to a well-

posed groundwater flow problem. The user is referred to Strack[1989] or Haitjema[1995] for a more complete description of the method.

1.2 Bluebird Scope and Current Capabilities

The Bluebird library currently has the ability to model a wide range of 2-Dimensional hydrogeologic features with various geometries and boundary conditions. The numerical solutions for each of these hydrogeologic features are superimposed for a complete description of a saturated groundwater flow field. A complete list of available elements is shown in Figure 1.1(A)

Figure 1.1(A): **Listing of available Elements in Bluebird**

Point Elements:	Discharge-Specified Well
	Head-Specified Well
	Drying Extraction Well
	Point Dipole
	Point Vortex
Circular Elements:	Circular Inhomogeneity in Conductivity
	Head-Specified Circular Lake
Linear Elements:	Discharge-Specified Horizontal Wells
Polylinear Elements:	Head Specified Rivers
	Resistance Specified Rivers
	Drains (Thin high-conductivity inhomogeneities)
	Leaky Walls (Thin low-conductivity inhomogeneities)
	Discharge Specified Rivers
	Normal Discharge Specified Boundaries
Polygonal Elements:	Polygonal inhomogeneities in conductivity
	Polygonal inhomogeneities in base and/or thickness
	Multi-quadric Area Sinks
	Multi-Quadric Inhomogeneity in Aquitard Conductance

Each of these element "objects" in the Bluebird library is uniquely associated with their own data members and functionality.

In the analytic element method, building a model corresponds to specifying important hydrologic features in an aquifer and solving for their coefficients so that they each meet their respective boundary conditions. In Bluebird, each of these features is associated with an *instance* of its class in the Bluebird library. For example, when you input a river into the model (along with all of its geometric information and specified heads), an instance of the class "CRiver" is created. The instance stores all of this geometric information, and is additionally associated with all of the operations you may perform upon a river: requesting information from it, modifying its properties, or ordering it to solve for it's own coefficients. Thus, each hydrogeologic feature ("element") is represented by a single instance of an object class in the Bluebird library, which can be modified, queried, or directed to perform some action.

In addition to the elements available in the Bluebird library, there is a set of "element containers", a set of global drivers, a collection of auxiliary functions, and a set of complementary object classes that simplify the design and modification of AEM models. The ***element containers*** are hydrogeologic constructs such as layers, aquifers, and aquitards. These contain all of the hydrogeologic features in the model and act as input and output parameters for the global drivers. The ***global drivers*** (such as Parse, Grid, Track, and WriteOutput) are routines that read input files, grid model output, and track particles along flow paths. The ***complementary object classes***, such as CParticle, CPathline, and CFlowNode, and the ***auxiliary functions***, such as optimized polynomial routines, aid in these actions. All of this functionality is embedded within a highly formalized design framework.

Design Assumptions

There are certain assumptions involved with the building of the Bluebird engine, most of which are associated with either the iterative algorithm or the assumptions inherent in traditional 2-Dimensional analytic element modeling. Some of the primary assumptions are discussed below:

2-Dimensional: The structure and organization of Bluebird is such that most information (such as potential or head) is only available at some (x, y) point (stored as a complex coordinate, $z=x + iy$) in some layer (L) of the model. Though locally 3-Dimensional "elements" or zones may be built into the framework of Bluebird (as in Haitjema [1985]), zones of 2-D and 3-D solutions may not overlap-they must be nested. Mathematically, this is due to the inherent assumptions of the Dupuit-Forcheimer assumption, but is functionally reinforced by the current structure of the program procedures.

Multi-Layer: While integration of fully 3-Dimensional models is somewhat outside the scope of the Bluebird engine, pseudo-3D models are explicitly accounted for. These models are comprised of layers connected by leakage through aquitards. While the multi-layer formulation is still somewhat incomplete, the structure of the library and associated engine fully accounts for multiple 2-Dimensional layers. The details behind this implementation are discussed later in this document.

Steady-State: Though the current implementation of Bluebird is fully steady state (there are no transient elements), the structure of most function calls accounts for time, t. In addition, the global solving, gridding, and tracking routines, are organized to account for a transient model. This, at some time in the future, may allow for two potential forms of transient models:

- Zones of transience nested within a steady state model (such that requests may be made for velocities or heads at any point in space time (x, y, t)) or
- Models that are analytic in space/finite difference in time. This type of model would require particle tracking and gridding to occur during the solution. An

alternative, to store information throughout the time history of the model, is operationally expensive, and not designed for.

Thus, while the model is currently limited to steady state considerations, later improvements may include transience, with little additional effort. The exception to this is the implementation of transient aquifer property values (i.e. conductivity, base, thickness), which would require rather dramatic restructuring.

Minimal communication between elements:

Bluebird was designed in such a way that the implementation of a new element or new element geometric type would require little or no modification of the rest of the code. One of the methods of minimizing the modification effort is to reduce element coupling. Elements have no knowledge of anything but the layer and Superblock that own them. Likewise, the layer and superblock know nothing about the type of elements they possess except that they have certain capabilities and features associated with all elements.

Iterative/Explicit Model: While the formulation of the Bluebird Object model was designed with the iterative solution algorithm in mind, an explicit solver (for use within a single layer) has been developed. However, the explicit solver is not yet operational for all of the element types.

Analytic Solution only: Generally, analytic element models are built via superposition of analytic solutions, which are infinite in domain. Future additions to the method may include use of local finite element, finite difference, or boundary element solutions to the governing equations, which may also be superimposed atop or nested within the analytic solutions. The general formulation of Bluebird classes and routines in no way impedes such an innovation.

1.3 Object-Orientation and AEM

The structure of analytic element models is inherently object-oriented.

Analytic Element Models, by their very nature, lend themselves to an object framework based on elements as the atomic classes. Elements each have their own geometric data (coordinates, shape), their own mathematical data (coefficients), and their own functionality (harmonic functions associated with the element). This alone suggests a high capacity for encapsulation of information and a high degree of connectivity between a single element's data. However, much of this information is similarly structured—inhomogeneities in conductivity and cutoff walls have identical functional form. Likewise, both a pumping well and circular lake provide similar functionality (i.e. provide discharge potential at a point). Thus, due to a clear hierarchy of classification, it is apparent that an inheritance-based architecture for these element objects is an intrinsic property of the analytic element method itself.

Booch [1994], suggests that the quality of an object abstraction may be measured by the following metrics:

- Coupling
- Cohesion
- Sufficiency
- Completeness
- Primitiveness

Coupling

Coupling is defined as the measure of strength of association between one object and another. Weak coupling is desired, because it is easier to understand and modify code for an object if it is only mildly interrelated with other object classes. The analytic element method, with hydrogeologic features as its classes, allows for minimal coupling—each element needs only knowledge of the layer within which it resides and the Superblock which owns it. An element may be entirely ignorant of all other elements in the model (or even know that others exist!). Likewise, a layer in the model needs to know nothing about the specific geometry or form of its internal elements, only to have the ability to request information from them, such as the discharge potential at a point. *An object-oriented formulation with elements as the atomic classes is weakly coupled, as desired.*

Cohesion

Cohesion is a measure of the degree of connectivity among the parts of a single object (or object class). Analytic elements are highly cohesive as an object base class, because they all provide very similar functionality (once again, potential at a point). Because of the very similar behavior shared by elements, the semantics of an object-oriented implementation may embrace the behavior of an analytic element, an entire analytic element, and nothing but an analytic element. *An object-oriented formulation with an abstraction of "Analytic Element" as the base class is highly cohesive, as desired.*

Sufficiency and Completeness

Sufficiency and completeness are metrics of a *specific implementation* of object-oriented design. Sufficiency requires a *minimal* interface for interaction between instances (i.e. interaction between the layer and an element). Completeness implies that *all* aspects of an abstraction are included in the interface. Thus, an analytic element abstract base class without an interface for "GetDischargePotential" would be insufficient and incomplete. *The analytic element method suggests an interface format that is both sufficient and complete, if only because certain information is* required *for a complete model to operate*.

Primitiveness

Primitiveness is a measure of the simplicity of a class's interface. For example, providing a function interface that provides the discharge at three given points is not primitive, because an external routine which calls GetDischarge(x, y) can be called three times. Once again, primitiveness is a function of the specific design implementation. It is important to note that the Iterative solution method is significantly more primitive than

the explicit method, which requires non-primitive function calls to build the matrix of equations.

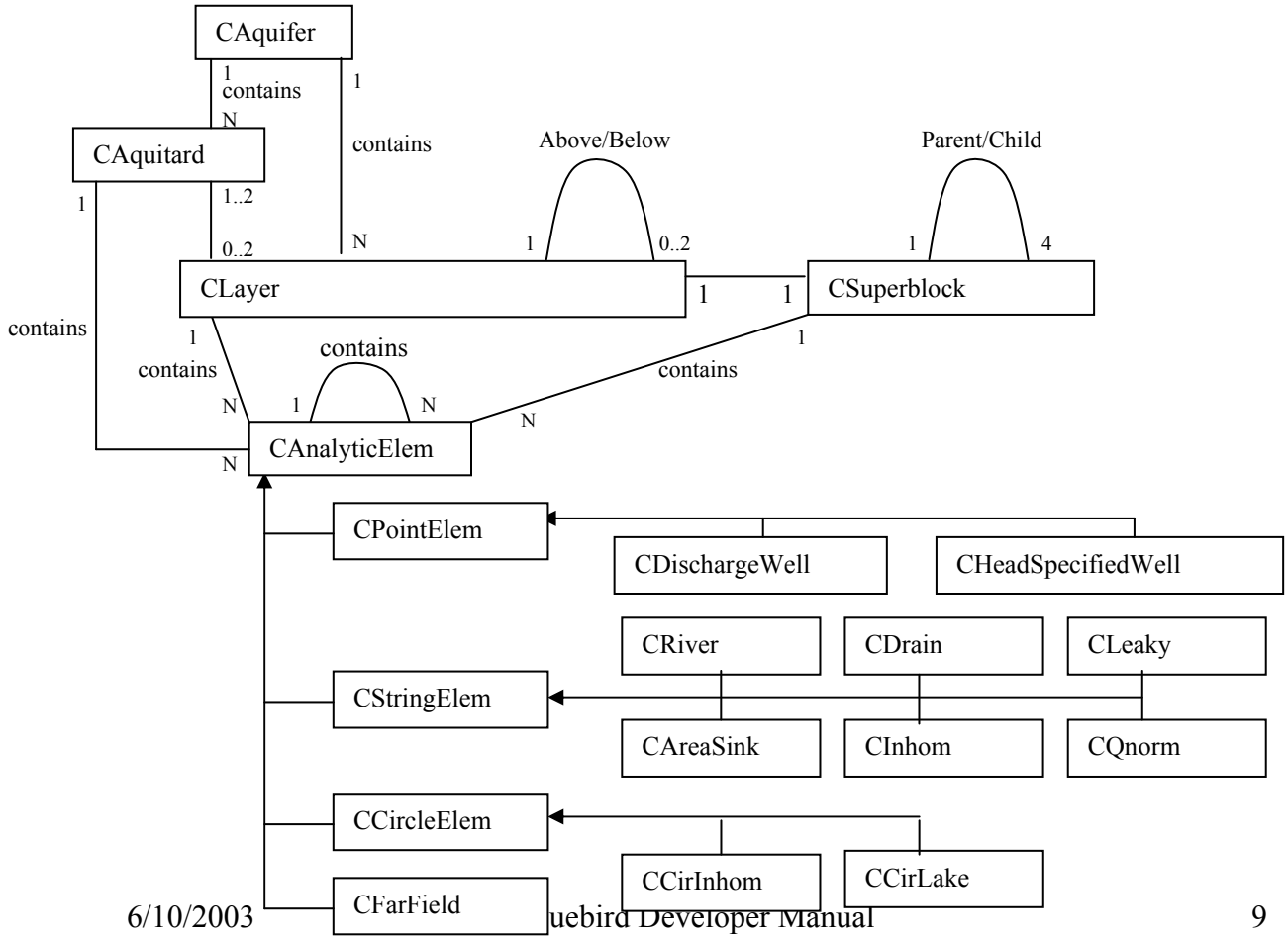
2.0 Code Organization

2.1 Overview

Generally, the Bluebird library is composed of a (1) set of analytic element classes (2) a set of container classes and (3) a set of global functions that operate upon instances of these classes. While exceptions exist to this generalization, the substance of the library lay in this basic structure. The element and container classes (the former of which is based upon a hierarchy of inheritance) are joined by two additional sets of complementary object classes (Property zone classes and miscellaneous classes). These four types of classes encapsulate the entire object-based library:

- *Container classes* - Objects that contain elements or groups of elements,
- *Element Classes* - Objects which are elements themselves, inheriting from the master class CAnalyticElem,
- *Property Zone Classes* - Objects that contain property information for a geographic region
- *Miscellaneous Classes* - Objects that interact with elements or represent some other type of hydrogeologic feature, such as a parcel of water or location of surface water communication.

The architecture for the container and element classes are shown in figure 2.1a. The element architecture is shown as a class (or inheritance) structure, whereas the container architecture is displayed as an object-relational structure.

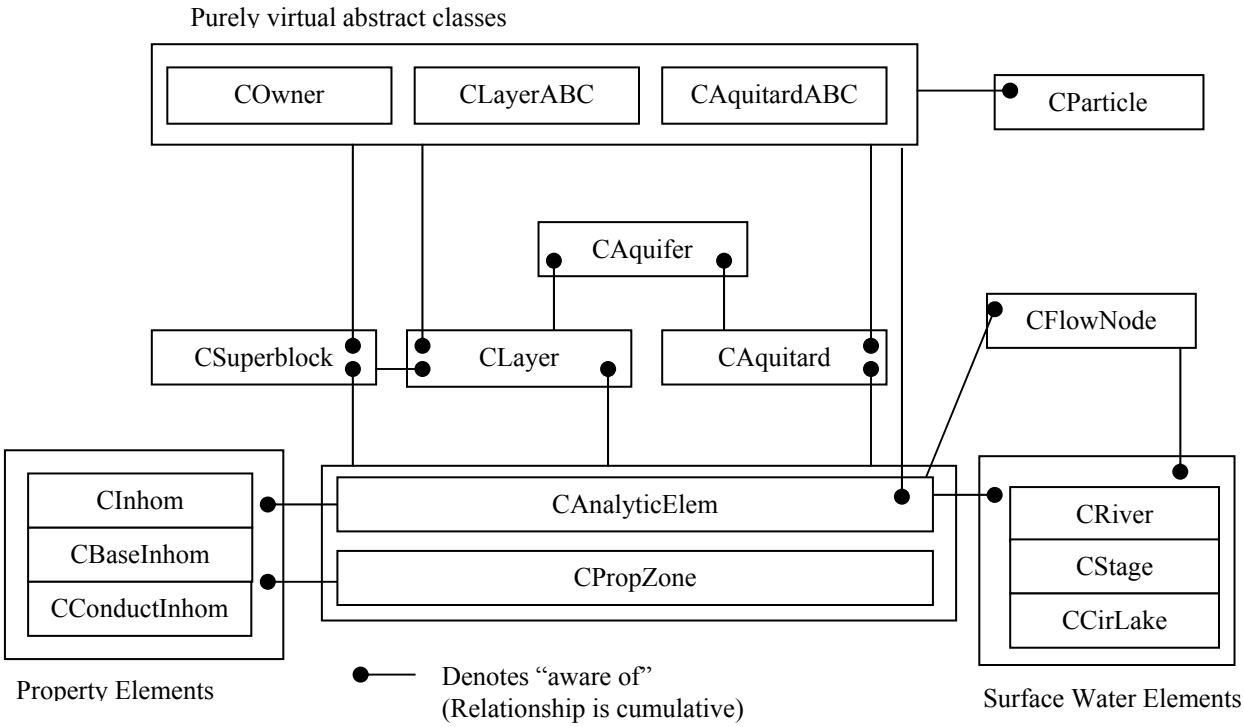


In addition to the object classes, there is a library of functions that operate outside of the formal class structure. These include general mathematical/boolean functions (min, max, Laurent series, Linear algebraic solvers, and string manipulation functions) and master functions for gridding and tracking using model results or processing input and output. An index of the general mathematical functions is available in appendix C. However, the primary driver functions for the library consist of:

- ***Main()*** – the primary driver function for the engine. Main() calls all of the other major functions.
- ***Parse()***-the parse routine parses the input file ("split.dat") and constructs the model domain, including all of the Layers, elements, particles, and auxiliary structures.
- ***Grid()***-The Grid routine grids (creates a *.grd file of) the head, stream function, leakage, Qx, and Qy in a specific layer at a specific time.
- ***BuildMatrix()***-The BuildMatrix Routine assembles a fully explicit matrix for a set of elements, then solves the system of equations for all of the element coefficients.

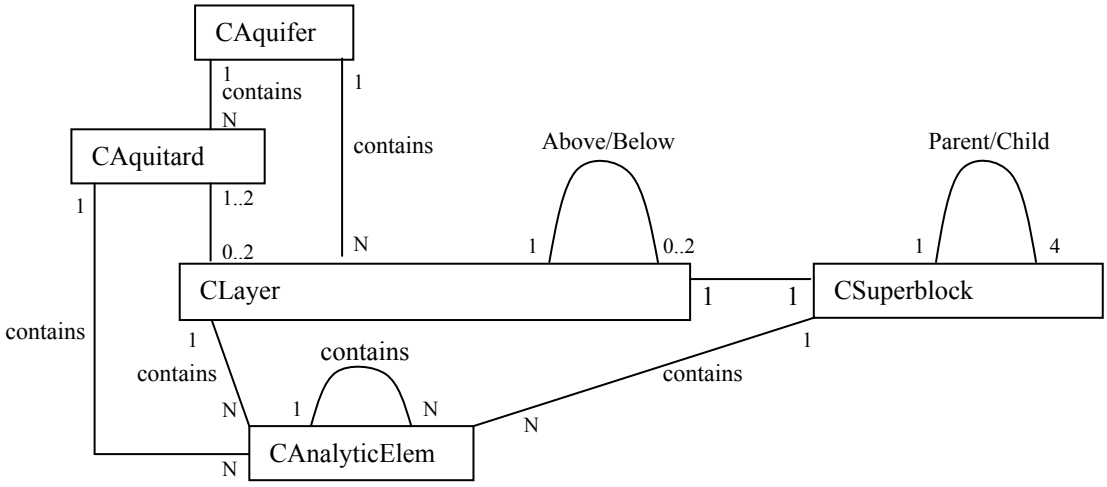
Awareness

The structure of the Bluebird library is such that each object has access only to the information it needs from other objects. Figure 2.1.2 represents the "awareness" structure of the Bluebird object library.



2.2 Container Classes

Container Classes are those classes that contain elements (or, equally, contain hydrogeologic features). There are natural container classes (such as aquifer) and mathematical container classes (such as Superblocks). The container organization for the Bluebird library is shown in fig 2.2a. Most of these containment relationships are connected via the use of c++ pointers. For example, a layer contains multiple elements, represented as an array of pointers to analytic elements (pElemArray[]). Each element, in turn, has a pointer to the layer within which it resides (pLayer). This type of structure is repeated for all of the container classes, and insures two-way flow of information between the container and the contained. It is important to note that ***the element containers do not explicitly know the type of elements contained within them***.



As depicted in fig. 2.2a, the object containers in the Bluebird library consist of:

- CAnalyticElem- The base class for all of the analytic elements is also a general aggregate container of analytic elements. While most analytic elements have a size of zero, the aggregate instantiation of CAnalyticElem can contain a group of elements unlimited in size.
- CLayer- The primary container class for analytic elements, CLayer represents an aquifer layer in which the Dupuit-Forcheimer assumption is valid.
- CAquitard- The aquitard class represents an infinitely thin leaky layer between the aquifer layers represented by CLayer. Only vertical flow is allowed in this layer.
- CAquifer-The Aquifer acts as a master container of layers and aquitards.
- CSuperblock- The Superblock class is a mathematical construct which groups elements and simplifies their functional form at a distance for faster computational time. It also speeds the access of location-based information via its quad-tree structure.

2.2.1 The Layer Class: CLayer

The layer class is the primary organizational class in the Bluebird library. It stores all of the elements within it (via an array, pElemArray[]) and has explicit knowledge about all of its contents. It inherits from the purely virtual class CLayerABC, which provides all of the functionality accessible from particles or elements. An element knows nothing of its surroundings- a river is ignorant of an adjacent inhomogeneity, and does not know if it exists in a multi layer model. It simply knows that it is within an abstract layer (of type CLayerABC), and asks the layer for important information, such as the discharge along its boundaries. The Layer is a "black box" to each of the elements, providing a limited amount of information, such as conductivity or potential.

CLayer Data Structures

Static members (not unique to Layer instance)					
Name	Type	Description	Status	Accesor?	Manipulator?
MinLayerIterations	integer	minimum local solve iterations	private	N	
MaxLayerIterations	integer	maximum local solve iterations	private	N	SetSolveData()
LayerTolerance	double	maximum acceptable local tolerance	private	N	
Fresh	boolean	true if first iteration	public	N/A	N/A
Solved	boolean	true if iteration complete	public	N/A	N/A
Data members (unique for each layer instance)					
conductivity	double	background conductivity	private	GetCond()	
base_elev	double	background base elevation	private	GetBase()	SetValues(K,B,T,n)
Thickness	double	background aquifer thickness	private	GetThick()	
Porosity	double	background porosity	private	GetPoro()	
black_hole	complex	z location where log terms go to zero	private	BH()	SetBlackHole()
pMasterBlock	CSuperblock*	pointer to master superblock	private	GetMasterBlock()	SetMasterBlock()
size	integer	number of elements in Layer	private	Y	
pElemArray	CAnalyticElem*[]	array of pointers to all elements	private	GetElem(i)	
PFarField	CFarField*	pointer to far field element	private	N	
Condsize	integer	number of conductivity inhomogeneities	private	N	
Basesize	integer	number of base inhomogeneities	private	N	AddToLayer()
Porosize	integer	number of porosity inhomogeneities	private	N	
pCondZoneArray	CPropZone*[]	array of conductivity inhomogeneity zones	private	N	
pBaseZoneArray	CPropZone*[]	array of base inhomogeneity zones	private	N	
pPoroZoneArray	CPropZone*[]	array of porosity inhomogeneity zones	private	N	
Level	integer	vertical location within aquifer system	private	Y	N
pLayerAbove	CAquitard*	pointer to leaky layer above	private	N	SetLevelAbove()
pLayerBeneath	CAquitard*	pointer to leaky layer below	private	N	SetLevelBelow()
DeltaPot	double	range of potential values in layer (Potmax-Potmin)	private	GetDeltaPot()	CalculateDeltaPot()
Extents	window	extents of modeling domain	private	GetExtents()	ChangeExtents(z)

CLayer Functions

Name	Input Parameters	Outputs	Type	Actions Performed
*GetHead	z(complex),t(double)	head(double)	double	Returns locally referenced head
*GetHeadAndPotential	z(complex),t(double)	omega(complex),head (double)	void	Returns local head and potential
*GetDischargePotential	z(complex),t(double)	omega(complex)	complex	Returns Discharge potential, Omega
*GetW	z(complex),t(double)	W(cmplex)	complex	Returns Complex Discharge Vector
GetNetDischarge	t(double)	Q(double)	double	Returns Net Outflux of water from domain
*v	z(complex),t(double)	velocity(complex)	complex	Returns darcian velocity
IterativeSolve	t(double),PROGRES S(output stream)	maxobjective(double), maxchange(double)	void	Solves all elements in domain iteratively
WriteItself	sol (output stream)		void	Orders its elements to write their coefficients to a solution file
WriteOutput			void	Orders its elements to write their output
Centroid		centroid of domain(cmplex)	complex	Returns the centroid of the domain

*Indicates inheritance from the abstract CLayer class, CLayerABC. These are the only functions accessible by the elements, particles, or gridding routines.

Use of CLayer

An instance of CLayer represents a purely 2-Dimensional infinite domain in which the Dupuit-Forcheimer assumption is valid everywhere. For a single layer model, all of the elements will reside within the single instance of the layer. Requests from the global drivers (such as Grid) for potential, flow, or net discharge are generally made through a pointer to a layer (i.e. the gridding routine asks the layer for potential and head at a set of points in order to create a .grd file of heads and stream function). Additionally, every element has a pointer to the layer in which it resides. Each element uses the layer as a "black box" to obtain information about the element's surroundings (i.e. Ω, W, conductivity, base elevation, or thickness).

2.2.2 The Aquitard Class: CAquitard

The aquitard, or leaky layer, class represents a resistant layer between aquifer layers. Whereas the Dupuit-Forcheimer assumption controls flow in the layers, in an aquitard, only vertical flux of water is permitted.

Use of CAquitard

The aquitard class is used only for multi-layer models. It has a set of elements associated with it (different formulations of area, line, and point sinks) that represent flux from one aquifer layer to another based upon the vertical flux relationship:

$$\gamma = \frac{k}{b}\phi_{upper} - \phi_{lower}$$

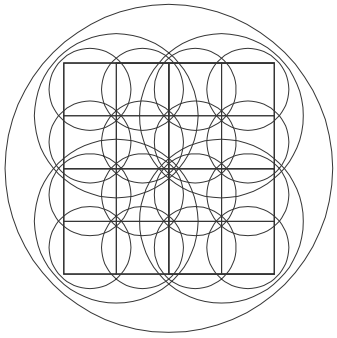
Where k is the conductivity of the aquitard, b is the thickness of the aquitard, and c is the conductance (c=k/b). This flux would be positive for one layer (an area sink) and negative for the other (area source). The aquitard class currently has only one type of element: inhomogeneities in conductance (conductivity and/or thickness), which has knowledge of the abstract CAquitardABC in which it resides. The default conductance of an aquitard (and the conductance at infinity) is zero, allowing no flux of water.

2.2.3 The Aquifer Class: CAquifer

The aquifer class represents the ***master container***: it contains all of the layers and leaky layers, which in turn contain all of the analytic elements. Calls to solve the domain are directed to the single instance of CAquifer. This instance sifts through all of its layers then all of its leaky layers, instructing them to solve themselves. Aside from acting as a master container of layers and aquitards, the Aquifer has very little functionality or data. It is envisioned that at some point, multiple adjacent aquifers may be "tiled" together, each with different vertical layering of its levels.

2.2.4 The Superblock Class: CSuperblock

Superblocks are a method of mathematical representation whereby groups of analytic elements may be represented at a distance by a single Laurent series and a singularity (well). The Bluebird library's implementation of nested superblocks takes advantage of this representation for faster numerical calculation. There is a set of nested superblocks in each layer that "own" a set of elements within their respective "domains". Inside this domain (a simple circular radius), the explicit element functions are used to represent potential. Outside this domain, a Laurent series is used to represent the grouping of element functions. Aside from the layer, Superblocks are the only construct that an analytic element has explicit knowledge of. Every time an element solves for its own coefficients, it informs it's superblock owner, so that it may update its Laurent coefficients and its parent blocks coefficients, and so on. A figure of the geometric structure of nested superblocks is in figure 2.2.4a.



2.3 Element Classes & Subclasses

The element classes are the primary object class in the Bluebird library, each instance representing a hydrogeologic feature such as a river, inhomogeneity, or zone of recharge.

2.3.1 The Generic Analytic Element Class: CAnalyticElem

The analytic element class is the master class for all analytic elements, regardless of geometry or specific boundary condition type. All elements are an instance of the CAnalyticElem class, and gain additional functionality via their subclasses, which are based upon geometry and boundary condition type.

The Inheritance Structure

The concept of inheritance in object-oriented design allows for sharing of data and functionality from one class to another without loss of functionality. It may be viewed as an "is-an" relationship: a river is an element, and therefore the CRiver class inherits from the CAnalyticElem class. In the Bluebird library, there is a four-tier hierarchy of inheritance. This inheritance structure is shown via example in fig 2.3.1a and shown abstractly in fig 2.3.1b

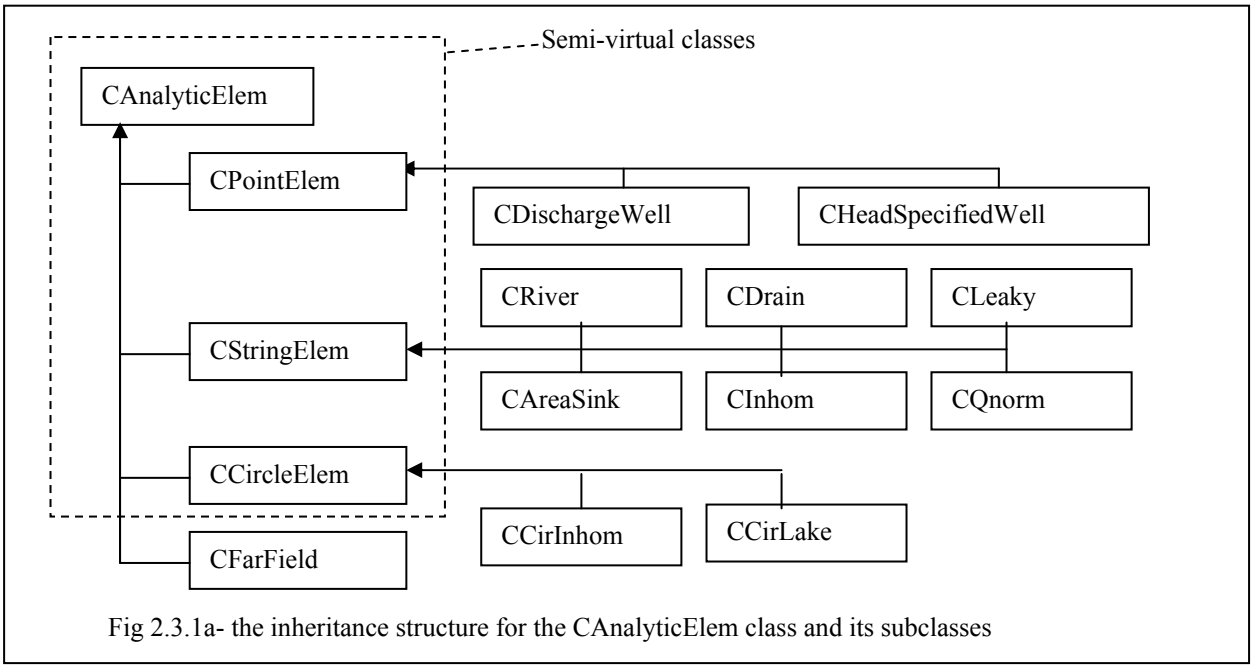


Fig 2.3.1a- the inheritance structure for the CAnalyticElem class and its subclasses

The Geometry type subclasses (i.e. CStringElem, CPointElem) contain the general potential function for all elements of that geometric type. They also contain

generalized information such as the coefficient structure and the geometric data. With the exception of CFarField, an instance of a Geometric Type Element still lacks essential information to solve for its own coefficients. The Boundary condition subclasses contain the specific BC data (such as conductivity or elevation of the element) that allow an element to be operational. It is at this (the BC) level of abstraction that an element can solve for its own coefficients.

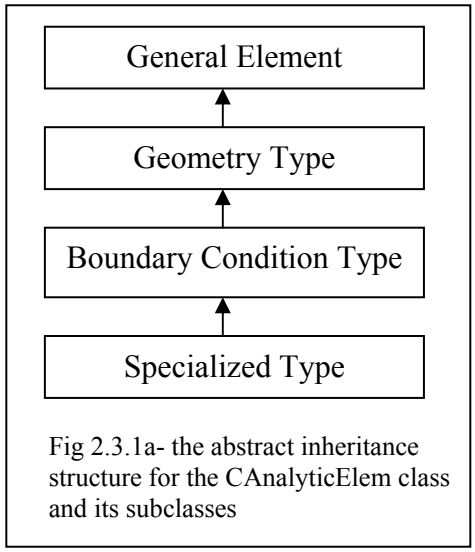


Fig 2.3.1a- the abstract inheritance structure for the CAnalyticElem class and its subclasses

Virtual Status

CAnalyticElem is what is called a "virtual" class, because an instance of CAnalyticElem has no real power or functionality without inheriting this power from additional child classes. For example, a river segment, which inherits from the classes CAnalyticElem, CStringElem, and CRiver, has associated geometry (from CStringElem) and associated boundary conditions (from CRiver). The additional functionality inherited from these subclasses allows it to have coefficients and a mathematical form, which allows it to provide potential or flux at a point. An instance of CAnalyticElem alone has none of this functionality, and its GetDischargePotential() or GetW() routine will always return 0.0+i0.0.

Element Awareness

Each element has no explicit knowledge about its surroundings. This allows for object-oriented code development where minimal modification of code is necessary to build new elements, containers, or functionality (see chapter 4 for details on modification). ***The knowledge of each element is limited to the layer in which it resides and the superblock(s) which "own" it.*** This "awareness" is represented via the pointers pLayer (a pointer to the abstract layer of class CLayerABC) and pBlock (a pointer to the block of abstract class COwner). In addition, each element has knowledge of its element ID within the superblock (allowing the superblock to update its coefficients correctly every time the element is solved for).

Data Members

Because CAnalyticElem is a predominantly virtual class, it has very few (and very general) data members. The data is listed below:

Data members (unique for each element instance)					
elemID	integer	identification num of element	private	GetID	
type	elemType	enumerated type of element	private	GetType	Constructor
name	string	name of element	private	GetName	
size	integer	number of elements in container (if an aggregate)	private	GetSize	AddToContainer
pElemArray	CAnalyticElem*[]	array of pointers to subelements (if an aggregate)	private	GetAllElems	
pLayer	CLayer*	Pointer to layer in which the element resides	private	N	Constructor

pBlock	CSuperblock*	pointer to superblock which "owns' the element	private	N	SetBlockOwner
myBlockID	integer	element's superblock ID number	private	N	

Member Functions

There are certain features and behaviors that all Analytic Elements share. The virtual functionality of the CAnalyticElem class allows this functionality to be accessed without explicit knowledge of the type of element (i.e. well or river). The following is a list of the functionality available from *all* elements:

Name	Input Parameters	Outputs	Type	Actions Performed
GetDischargePotential	z(complex),t(double)	omega(complex)	complex	Returns Discharge potential, Omega
GetW	z(complex),t(double)	W(cmplex)	complex	Returns Complex Discharge Vector
GetNetDischarge	t(double)	Q(double)	double	Returns Net Outflux of water from domain
SolveItself	t(double)	maxobjective(double), maxchange(double)	void	Solves for coefficients of element(or subelements of an aggregate)
WriteItself	sol (output stream)		void	Write its coefficients to a solution file
ReadItself	sol (input stream), solfiletype	success(boolean)	bool	Reads its coeffifients from a solution file
WriteOutput			void	Writes its output to predefined files
UpdateBlock			void	Tells owning superblock to update it's coefficients
Centroid		centroid of element(cmplex)	complex	Returns the centroid of the domain
IsInside	z(complex)	boolean	boolean	returns true if a point is inside the element
IsInSquare	zc(complex),width(double)	boolean	boolean	returns true if the element is inside the given square
IsInCircle	zc(complex), radius(double)	boolean	boolean	returns true if the element is inside the given circle
SharesNode	znode(complex)	boolean	boolean	returns trure if the element has a node at the given point

Modeling/Behavioral Functionality

- GetDischargePotential (complex z, double t)
- GetW(complex z, double t)
- GetNetDischarge(double t)
- SolveItself(double &change, double &objective, double t).

The Discharge Potential ($\Omega_e(x,y,t)$):

All elements have a mathematical function which represents an element's contribution to the discharge potential $\Omega=\Phi+i\Psi$ [L^3/T] at a point within its layer. While the form of this function is dependent upon the specific geometry and behavior of the hydrogeologic feature, ALL elements have the ability to return the value of Ω at a point in time and space. Therefore, CAnalyticElem provides the virtual functionality for this function: ***GetDischargePotential (complex z, double t)***. The input parameter z is the complex 2-D location (within the layer) and the parameter t is time. The functional form of this routine is built within the geometry subclass (i.e. all circles or all point elements share the same form of *GetDischargePotential* function). The specific functional form for each geometry type is given in the discussion of Geometric element subclasses (sect. 2.3.2)

The Complex Discharge ($W_e(x,y,t)$):

Like discharge potential, all elements have the ability to provide the derivative of the discharge potential, the complex discharge, $W=Q_x + iQ_y$ [L^2/T]. Similarly, the functional form is provided by the geometry of the element, but the function is available regardless of geometry. Therefore, all instances of CAnalyticElem can respond to the function ***GetW(complex z, double t)***. The input parameter z is the complex 2-D location (within the layer) and the parameter t is time. The functional form of this routine is built within the geometry subclass (i.e. all circles or all point elements share the same GetW function)

Net Discharge ($Q_e(t)$):

Though not all elements add or remove water from the domain, it is important to be able to access this information without explicit knowledge of an element's form. Therefore the virtual function ***GetNetDischarge(double t)*** returns the volumetric flux of water [L^3/T] removed from the layer by the element at time t. The functional form of this routine is built within the geometry subclass (i.e. all circles or all point elements share the same GetNetDischarge function)

Solve Itself:

One of the primary assumptions in the iterative method for solution of analytic element models is that an element may solve for its own coefficients with limited information along its borders. The iterative solution process consists of looping through the list of elements and having each element solve for its own coefficients based upon the most recent global solution. Therefore, every instance of CAnalyticElem has the functionality ***SolveItself(double &change, double &objective, double t)***. For any time t, the function solves for its coefficients and returns the values change and objective, which are the degree of coefficient change and the element's ability to meet its boundary conditions, respectively. The functional form of this routine is built within the boundary condition subclass (i.e. all circular lakes have the same SolveItself function, which is different from the circular inhomogeneity SolveItself function)

Input/Output Functionality

- void WriteItself(ofstream &solfile)
- void ReadItself(ifstream &solfile, solfiletype ty)
- void WriteOutput(double t)

Write Itself:

Each element has the capacity to write it's coefficient information to a solution file (to be stored for future use). The function call used is ***WriteItself(ofstream &solfile)***. The input parameter, solfile, is an output stream that points to the solution file.

Read Itself:

In addition to writing its solution, an element must be able to read its solution from an input solution file. The function call is ***ReadItself(ifstream &solfile, solfiletype ty)***. The first parameter is an input stream pointing to the input solution file. The second

parameter is the type of input solution file (so that elements can read their input from a different program's (i.e. SPLITS) solution file).

Write Output:

Different elements may provide different output. Rivers can provide extraction along their length, most elements can provide information about the errors along their boundaries. All of these output features are placed in the subroutine ***WriteOutput(double t)***.

General Geometric Functionality

- Centroid()
- IsInside(complex z)
- IsInSquare(complex zcen, double width)
- IsInCircle(complex zcen, double radius)
- PartInCircle(complex zcen, double radius)
- SharesNode(complex znode)

Centroid:

The routine ***Centroid()*** returns the center point of the element

Is Inside:

The routine ***IsInside(complex z)*** returns true if the point z is in the element, false otherwise.

Is In Square:

The routine ***IsInSquare(complex zcen, double width)*** returns true if the element is completely within the square defined by zcen (the center of the square) and width (the length of a side of the square). The square is oriented with the Cartesian coordinate system.

Is In Circle:

The routine ***IsInCircle(complex zcen, double radius)*** returns true if the element is completely within the circle defined by zcen (the center of the circle) and radius (the radius of the circle).

Part In Circle:

The routine ***PartInCircle(complex zcen, double radius)*** returns true if the element is *partially* within the circle defined by zcen (the center of the circle) and radius (the radius of the circle).

Shares Node:

The routine ***SharesNode(complex znode)*** returns true if the element has a node (i.e. corner point or singularity) at the point znode, false otherwise. Rarely used.

2.3.2 Geometric Element Subclasses

The Geometry type subclasses (CStringElem, CPointElem, CFarField, and CCircleElem) contain the general potential function for all elements of that geometric type. They also contain generalized information such as the coefficient structure and the geometric data. With the exception of the *SolveItself*, *GetMatrixBuildInfo*, and *WriteOutput* routines, all of the virtual CAnalyticElem functions are given a body at this level of inheritance. This insures minimal overhead in function calls and minimal reuse of code[1]. All of the virtual geometric queries from CAnalyticElem are the same for elements of the same geometries.

The Point Element Class: CPointElem

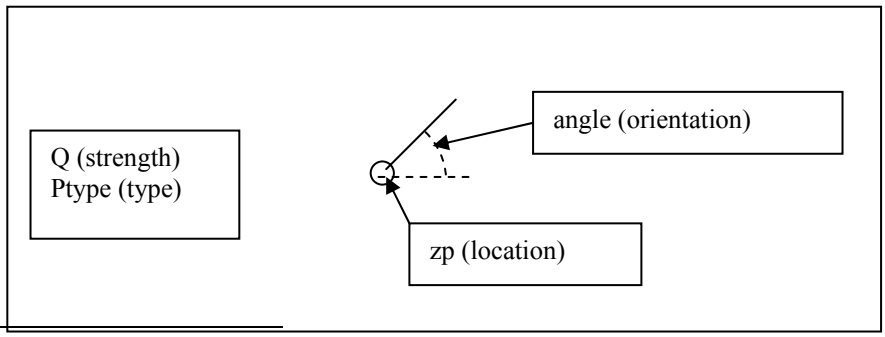
The CPointElem class currently includes

- head-specified wells,
- discharge- specified wells,
- discharge- specified wells which will not over pump (drying wells),
- point dipoles, and
- point vortices.

The general class contains a set of private data members, which generalize to all point elements, and provides content to the virtual geometric and modeling functions from CAnalyticElem, with the exception of *SolveItself* and *GetMatrixBuildInfo*. These functions have content only at the boundary-condition level of inheritance.

CPointElem Data Members

Data members (unique for each CPointElem instance)					
Name	Type	Description	Status	Accessor?	Manipulator?
zp	complex	the complex location, z_p	Private	GetZ()	Constructor
Ptype	complex	enumerated point type (well, vortex, or dipole)	Private	N	
Q	integer	single degree of freedom, the strength coefficient Q	Private	GetNetDischarge()	
angle	integer	orientation, q, in terms of radians from the positive x-axis	Private	getOrientation()	



[1] Since the form of the Ωand W functions are so similar for elements of similar geometry, it makes sense to encapsulate these functions at the level of geometric differences, rather than repeat this code in a similar form for each string element or point element.

CPointElem Member Functions

Get Potential, Get W, and Get Net Discharge

The general expression for the complex potential and complex discharge from a point element is given as:

$$\Omega_e(z) = \frac{Q}{2\pi}\left(\delta_1 \ln\left(\frac{z - z_p}{z_{bh} - z_p}\right) - \delta_2 \frac{e^{i\theta}}{z - z_p}\right)$$

$$W_e(z) = -\frac{Q}{2\pi}\left(\delta_1 \frac{1}{z - z_p} - \delta_2 \frac{e^{i\theta}}{(z - z_p)^2}\right)$$

$$Q_e = \Re(\delta_1 Q)$$

$$\text{where } \delta_1 = \begin{cases} 1 & (well) \\ -i & (vortex) \\ 0 & (dipole) \end{cases} \quad and \quad \delta_1 = \begin{cases} 0 & (well\ or\ vortex) \\ 1 & (dipole) \end{cases}$$

The Circular Element Class: CCircleElem

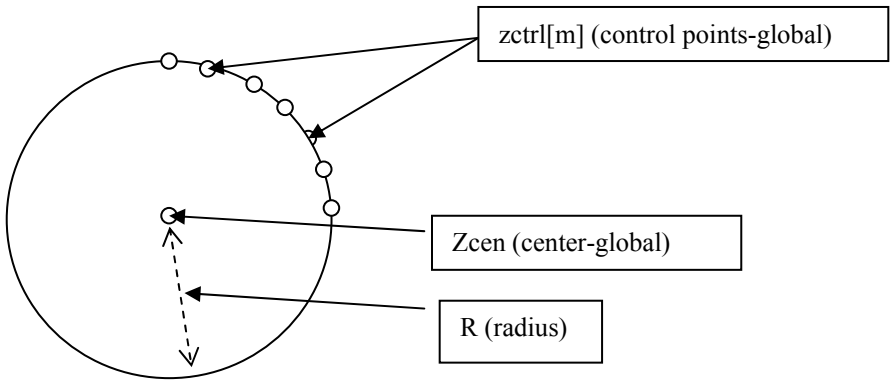
The CCircleElem class currently includes head-specified lakes and circular inhomogeneities in conductivity. The general class contains a set of private data members, which generalize to all circular elements, and provides content to the virtual geometric and modeling functions from CAnalyticElem, with the exception of *SolveItself* and *GetMatrixBuildInfo*. These functions have content only at the boundary-condition level of inheritance.

CCircleElem Data Members

Data members (unique for each CircularElem instance)

Name	Type	Description	Status	Accessor?	Manipulator?
OutCoeff[]	complex	An array of complex Laurent Series coefficients which represent potential outside the circle	private	N	SetCoeff()
InCoeff[]	complex	An array of complex Taylor Series coefficients which represent potential inside the circle	private	N	
zcen	integer	The complex center of the circular element.	private	GetZcen	Constructor
R	integer	The radius of the circular element	private	GetRadius()	
Q	integer	The discharge from the center of the circle.	private	GetNetDischarge	Constructor (Sometimes)
zctrl[]	complex	An array of the control points along the circular element's boundary	private	N	
ncircontrol	complex	The number of control points along the boundary	private	N	
order	double	The order of both the inner and outer power series (i.e. the number of OutCoeff and/or InCoeff)	private	GetDegreesOfFreedom()	Constructor
fold	integer	The over specification fold for the element (ncircontrol/order)	private	N	

CCircleElem Geometric Abstraction



CCircleElem Member Functions

Get Potential, Get W, and Get Net Discharge

The general expression for the complex potential and complex discharge from a general circular element is given as:

$$\Omega_e(z) = \delta\sum_{n=0}^{N} a_n Z^n + (1-\delta)\sum_{n=0}^{N} b_n Z^{-n} + \delta\frac{Q}{2\pi}\ln\left(\frac{z-z_c}{z_{bh}-z_c}\right)$$

$$W_e(z) = \delta\sum_{n=1}^{N} n a_n Z^{n-1} + (1-\delta)\sum_{n=1}^{N} -n b_n Z^{-n-1} + \delta\frac{Q}{2\pi}\frac{1}{z-z_c}$$

$$Q_e = Q$$

$$\text{where } Z = \frac{z-z_c}{R} \quad and \quad \delta = \begin{cases} 1 & Z > 1 \\ 0 & Z \le 1 \end{cases}$$

here a_n=OutCoeff[n] and b_n=InCoeff[n]. The values of these coefficients are dependent upon the specific boundary conditions of the element.

SolveItself

The functional form for circular elements allow for the coefficients (even in a high-order implementation) to be solved for individually. Thus, no additional generic function is required (as with string elements) to aid in the SolveItself routine.

The Polyline/Polygon Element Class: CStringElem

The CStringElem class currently includes a large group of elements:

- Head-Specified Rivers
- Extraction-Specified Rivers
- Inhomogeneities in Conductivity
- Inhomogeneities in Base
- Thin Inhomogeneities (Drains)
- Thin Inhomogeneities (Leaky Walls)
- Normal-Discharge Specified Elements
- Resistance Specified Rivers
- Area-Sinks
- Horizontal wells

The general class contains a set of private data members, which generalize to all polyline or polygon elements, and provides content to the virtual geometric and modeling functions from CAnalyticElem, with the exception of *SolveItself* and *GetMatrixBuildInfo*. These functions have content only at the boundary-condition level of inheritance.

CStringElem Data Members

Static members (not unique to String instance)					
Name	Type	Description	Status	Accessor?	Manipulator?
c[4][]	double	generic boundary condition coefficients (if non-constant within element)	private	N	
c1,c2,c3,c4	double	generic boundary condition coefficients (if constant within element)	private	N	SetConstraints()
ConstrntOn[3]	boolean	true if constraints on	private	N	
ConstrntVal[3]	double	values of constraints	private	N	
CD	double	true if iteration complete	private	N	
f[ ][ ]	double	g matrix (for chebyshev polynomials)	private	N	Prepare()
g[ ][ ]	double	f matrix (for chebyshev polynomials)	private	N	
rhs[ ]	double	right hand side terms for line element solve	private	N	SolveItself()
sol[ ]	double	solution vector from solution matrix	private	N	
A[ ][ ]	double	solution matrix coefficients	private	N	GenSolve()
b[ ]	double	rhs of explicit solution matrix (in GenSolve)	private	N	
unit[ ][ ]	double	matrix of unit influences at control points	private	N	BuildUnitMatrix()
myBlockIDs	integer	segment block ID numbers	private	N	

Data members (unique for each string instance)					
JumpCoeff[ ][ ]	double	2D array of jump function chebyshev coeff	private	N	SetSegCoeff()
FarFldCoeff[ ][ ]	double	2D array of far-field Laurent Series coeff	private	N	N
FForder[ ]	integer	order of Far Field Laurent Series	private	N	Constructor
order	integer	dynamic order of chebyshev jump polynomial	private	GetDegreesOfFreedom()	
eval_order[ ]	integer	order of chebyshev jump polynomial	private	N	N
ze[ ]	complex	array of polyline endpoints	private	GetZ(I), GetZArray()	
zctrl[ ][ ]	complex	2D array of control points (one 1D array per line)	private	GetSegUnitInfluences()	
X[ ]	double	array of control points in local coordinate system	private	N	Constructor
Nlines	integer	number of line segments in polyline	private	GetNumSegs()	
nlinecontrol	integer	number of control points per line in polyline	private	N	
fold	double	overspecification fold of element	private	N	
disabled[ ]	boolean	if segment i is disabled, disabled[i] is true	private	IsDisabled(i)	SetAsDisabled(i)
ltype	linetype	dipole, doublet, or linesink	private	N	Constructor
closed	boolean	true if polygon	private	N	
pBlocks[ ]	CSuperblock *	array of pointers to superblock owners (1 per segment)	private	N	SetBlockOwner()

CStringElem Non-Inherited Member Functions

Name	Input Parameters	Outputs	Type	Actions Performed
Private Member Functions				
SetConstraints	I(int),tmp1…tmp4 (double),Ctype(constrainttype)		void	Sets constraint parameters
SetConstraints	I(int),tmp1…tmp4,val1,val2(double)		void	Sets constraint parameters
SetConstraints	I(int),tmp1…tmp4,val (double),Ctype(constrainttype)		void	Sets constraint parameters
SetFarFieldCoeff	i(int)		void	Sets Far Field coefficients for segment i based upon jump coeff
BuildUnitMatrix	I(int), Ltype(linetype), multi_val_c, firsttwo(boolean)		void	Builds unit matrix, unit[ ][ ]
GenSolve	JumpCoeff [ ],relax(double),I(int), Ltype (linetype), multi_val_c, firsttwo (boolean)	objective, change(double)	void	Solves system of least squares equations
Clenshaw	n(int), x, coeff[ ](double)	value of clenshaw polynomial at pt x	void	calculates clenshaw polynomial
cClenshaw	n(int), z, coeff[ ](double)	value of clenshaw polynomial at pt z	void	calculates clenshaw polynomial
cClenshaw2	n(int), z, coeff[ ](complex)	value of clenshaw polynomial at pt z	complex	calculates clenshaw polynomial
Interpolate	size(int),v1,v2(double)	Array[ ](double)	void	interpolates values based upon fourier frequencies of a line
Interpolate	size(int),v1,v2(complex)	Array[ ](complex)	void	interpolates values based upon fourier frequencies of a line
GetOmJump	I(int),X,t(double)	Jump in complex potential (complex)	complex	Calculates Jump in complex potential
GetWJump	I(int),X,t(double)	Jump in complex discharge (complex)	complex	Calculates Jump in Complex Discharge
GetHeadJump	I(int),X,t(double)	Jump in head (real)	double	Calculates Jump in head
GetHeadGradJump	I(int),X,t(double)	Jump in head gradient (real)	double	Calculates Jump in head gradient
GetCumExtract	I(int),X,t(double)	Cumulative extraction along linesink	double	Calculates Cumulative Extraction along linesink
CleanStreamFunct	z(complex),t(double)	increase in stream function	double	forces branch cuts along polyline, instead of having branch cuts for every segment
Public Member Functions (not inherited)				
Prepare()			void	Prepares CD, f, and G matrices
Destroy()			void	Destroys CD, f, and G matrices

CStringElem Inherited Member Functions

Get Potential, Get W, and Get Net Discharge

The general expression for the complex potential and complex discharge from a general polyline element (with complex jump coefficients, a_n) is given as:

$$\Omega_e(z) = \sum_{i=1}^{NL}\left[\frac{1}{2\pi i}\left(\sum_{n=0}^{N} a_{i,n}\sum_{j=0}^{n} f(n,j,Z_i)T_n(Z_i)\right) - \frac{\delta}{2\pi}\left(\left(\sum_{n=0}^{N} a_{i,n}T_n(1)\right)\ln(Z-1) + \left(\sum_{n=0}^{N} a_{i,n}T_n(-1)\right)\ln(Z+1)\right)\right]$$

$$W_e(z) = \sum_{i=1}^{NL}\left[\frac{-2}{z_2 - z_1}\frac{1}{2\pi i}\left(\sum_{n=0}^{N} a_{i,n}\sum_{j=0}^{n} g(n,j,Z_i)T_n(Z_i)\right) + \frac{\delta}{2\pi}\left(\left(\sum_{n=0}^{N} a_{i,n}T_n(1)\right)\frac{1}{Z-1} - \left(\sum_{n=0}^{N} a_{i,n}T_n(-1)\right)\frac{1}{Z-1}\right)\right]$$

$$Q_e(z) = \sum_{i=1}^{NL}\left[\delta\sum_{n=0}^{N} a_{i,n}\left(T_n(1) - T_n(-1)\right)\right]$$

$$\text{where } Z_i = \frac{z - \frac{1}{2}(z_i^1 + z_i^2)}{\frac{1}{2}(z_i^1 - z_i^2)}$$

where δ=1 for a linesink, 0 otherwise. The definitions of these terms are fully explained in Janković [1997].

The jump coefficients here are purely real for linesinks and dipoles, purely imaginary for doublets. For storage simplification, only the non-zero part is stored in a array of type double. Note that outside of a circle around the element, a Laurent series must be used. This is to avoid numerical inaccuracies in the calculation of the complex Chebyshev polynomial $T_n(Z)$, where Z is large.

The exception to this formulation is the area sink, which has an additional term for the potential on its interior. The Area sink, therefore, has its own interpretation of the ***GetDischargePotential()***, ***GetW()***, and ***GetNetDischarge()*** functions.

The Far Field Element Class: CFarField

The Far Field element, which is a combination of the Global constant, C, and uniform flow, is given its own geometric subclass because of its unique infinite geometry- both uniform flow and the global constant are effective at every point in the domain. Therefore, it has its own interpretation of the geometric function calls and unique functions for discharge potential, Ω, and complex discharge, W. While the uniform flow rate is given, the contstant may be determined in one of two ways- either based upon specifying the head at a complex point, z_{ref}, or by specifying total net extraction from the domain.

CFarField Data Members

Data members (unique for each FarFieldElem instance)

Name	Type	Description	Status	Accessor?	Manipulator?
Qo	complex	The uniform flow rate [L^2/T] $Qo=Q_x+iQ_y$	private	getUnifFlow()	
Alpha	double	The angle of uniform flow (in radians from the positive x-axis)	private	N	Constructor
RefPoint	boolean	True if reference point condition, false if net extraction condition	private	IsRefPoint()	
zref	complex	Location of the reference point	private	GetZref()	SetZref()
RefHead	double	Specified head at the reference point	private	N	Constructor
NetExt	double	Specified Net Extraction	private	N	
Constant	double	The Global constant, C	private	N	SetCoeff()

CFarField Member Functions

Get Potential, Get W, and Get Net Discharge

The general expression for the complex potential, complex discharge and net flux from a general Far Field element is given as:

$$\Omega_e = -\overline{Q}_O (z - z_{bh}) + C$$
$$W_e = \overline{Q}_O$$
$$Q_e = 0$$

Solve Itself

Since the Far Field element is a non-virtual element, it has full functionality, including the ability to solve for its own degrees of freedom, which is the global constant. The boundary condition for the farField is one of two conditions:

$$\Phi(z_{ref}) = \Phi_{specified} \qquad \text{or}$$
$$\sum_{e=1}^{N} Q_e = 0$$

Due to numerical difficulties with the second (net-extraction) condition (met essentially via a root-finding algorithm), relaxation is used.

2.3.3 Boundary Condition Element Subclasses

All of the Boundary condition (BC) subclasses inherit from some geometric subclass (CPointElem, CCircleElem, or CStringElem). With most of the functionality encapsulated at the level of geometry, the BC subclasses need only a few element-specific functions, which are virtual at higher levels:

- Constructor
- SolveItself
- GetMatrixBuildInfo
- Parse
- WriteOutput

These 5 member functions (along with some manipulators and accessors) are generally the only functions which are specific to the boundary condition subtype. Most other functionality is purely geometry dependent.

As an example of a boundary condition-based subclass, the CInhom (polygonal inhomogeneity in hydraulic conductivity) will be used.

Constructor:

The constructor builds an instance of the particular type. As an example, the constructor call for CInhom is:

```
CInhom::CInhom (char         *Name,
                const CLayer *pLay,
                const cmplex *points,
                const double  cond,
                const int     ord,
                const double  OS,
                const int     NumOfLines)
```

This input is all that is needed to completely define a polygonal inhomogeneity element:

- The name (*Name) and layer pointer (pLay) are general to all analytic elements.
- The array of vertices (points[]), order (ord), number of line segments (NumOfLines), and over specification ratio (OS) are all member data of a polyline element, and completely describe the geometry of the element.
- The only additional data required is the conductivity (cond), which is the single data member unique to an inhomogeneity in conductivity.

Upon calling this constructor, the CAnalyticElem and CStringElem Constructors are also called. The constructor (and parent constructors) builds a complete instance of CInhom, which may be added to a layer and later used in calculation.

SolveItself:

The ***SolveItself(double &change, double &objective, double t)*** routine is the meat behind the Bluebird iterative solver. Based upon the most recent complex discharge potential (Ω) and complex discharge (W) along the elements borders, an element may solve for its own coefficients by meeting its associated boundary conditions. This is done by first obtaining current information about the state of the system along its borders. The discharge potential and/or complex discharge are obtained by asking the layer for information:

```
O=pLayer->GetDischargePotential(z,t);

W=pLayer->GetW(z,t);
```

In addition to obtaining the values for Ω and W, the element may need information regarding the local specified conductivity, thickness, or base along its borders. These function calls are also made via the pLayer pointer (remember, all of an element's outside information is from the layer):

```
K=pLayer->GetCond(z);

B=pLayer->GetBase(z);

T=pLayer->GetThick(z);
```

This is the only information required for most traditional analytic elements to solve for their coefficients. In the case of a polygonal inhomogeneity, only the conductivity, K, and Potential, Φ, along its sides, is required. This is because the boundary condition for a polygonal inhomogeneity is:

$$\Delta\Phi_{inh,j} = 2\left(\frac{k^+ - k^-}{k^+ + k^-}\right)\Phi_{\neq inh,j}$$

This insures continuity of head along the element's boundary. This boundary condition is purely a function of the interior conductivity, k^+ (the CInhom data member, `kin`), the exterior conductivity, k^-, (from the function call `pLayer->GetCond(z)`), and the potential from everything but the element segment j (from the function call `pLayer->GetDischargePotential(z,t)`). Therefore, with only the black box pLayer to guide the element, it can solve for its own coefficients, which are stored in the member data JumpCoeff[][] (a complete discussion of the mathematics is in Janković [1997]).

The mathematical routine for *SolveItself()* is different for each element geometry and boundary condition type[2]. However, the procedure is essentially the same for all elements:

[2] CInhom solves for its coefficients using the CStringElem module GenSolve, for the least-squares solution of a line element with a Chebyshev polynomial. Oftentimes, the coefficient solution method is similar enough for the same element geometry to encapsulate some general solve functions at the geometry level.

- Get information along the element borders (at control points) from the layer
- Using this information, solve for the element coefficients.
- If Superblocks are on, inform the parent superblock that there are new coefficients

Though the mathematics or the boundary conditions may be complex, this routine explains exactly how *all* element *SolveItself* routines work. The exception to this recipe is for "given" elements, which solve themselves only on the first iteration (and then, only to update their superblock coefficients). "Given" elements have behavior which completely independent of the local potential and discharge (i.e. leakage-specified area sinks, discharge-specified extraction wells).

GetMatrixBuildInfo:

The ***GetMatrixBuildInfo()*** routine is used for the globally or locally explicit solver. While the routine has (at this point) only been developed for the river and far field elements, its application with other elements is straightforward. The general formulation for the explicit system of equations is as follows:

$$\sum_{n=1}^{DOF_s} a_n \sum_{m=1}^{M_s} \overset{m}{u}_n \overset{m}{u}_s - \sum_{k=1}^{DOF_{\neq s}} a_k \sum_{m=1}^{M_s} \overset{m}{\alpha}_{sk} \overset{m}{u}_s = \sum_{j=1}^{N_g} \sum_{m=1}^{M_s} \overset{m}{\alpha}_{sj} \overset{m}{u}_s + \sum_{m=1}^{M_s} \overset{m}{\alpha}_s \overset{m}{u}_s \quad s = 1...DOF_{TOTAL}$$

where a_n are the element coefficients,

$\overset{m}{u}_n$ is the unit influence of a single coefficient at control point m, and

$\overset{m}{\alpha}_{sk}$ is the RHS term, which includes additional terms for the boundary condition associated with the element coefficient a_s.

DOF_s is the number of degrees of freedom associated with the element that owns coefficient s

$DOF_{\neq s}$ is the number of degrees of freedom associated with all elements which do not own element s

DOF_T is the total number of degrees of freedom

N_g is the number of given elements, and

M_s is the number of control points associated with the element that owns coefficient s

The general boundary condition unit influence of coefficient k upon coefficient s may be written out as:

$$\overset{m}{\alpha}_{sk} = \beta_{\Phi,s} \overset{m}{u}_{\Phi,k} + \beta_{Qx,s} \overset{m}{u}_{Qx,k} + \beta_{Qy,s} \overset{m}{u}_{Qy,k}$$

The GetMatrixBuildInfo() routine provides the information required from a single element to build the matrix by sending the following structure to the *MatrixBuild*() routine.

```
struct MatrixInfo{
      int    nctrl;
      cmplex zctrl  [maxcontrolpts];
      double elemrhs[maxcontrolpts];
      double unit   [maxDOF][maxcontrolpts];
      double phiCoeff;
      double QxCoeff;
      double QyCoeff;
};
```

Where nctrl is the number of element control points,
zctrl is the locations of these points,
elemrhs is the contribution of the element to the RHS, $\overset{m}{\alpha}$,
unit is a 2D array of unit influences, $\overset{m}{u}_n$, and
phiCoeff (β_Φ)
QxCoeff (β_{Qx})
QyCoeff (β_{Qy})
represent the required information from other elements at the boundary (e.g. for an inhomogeneity, $\beta_\Phi=2*(k^+-K^-)/(k^++k^-)$, $\beta_{Qx}=0$, and $\beta_{Qy}=0$).

Parse:

The parsing routine, ***Parse()***, accepts an input stream, line number (the line of the input file the stream is currently pointing to, the name of the element (which has already been parsed) and a pointer to the current layer being parsed (the layer in which the element resides). The routine builds the element based upon the parsed information and returns a pointer to the element. If an error occurs, the routine returns a NULL pointer or exits the program gracefully. For an inhomogeneity, the input file command is (as in Janković, [2000]):

"Inhomogeneity" [Inhom name]	1
[conductivity]	2
[x1] [y1]	3
[x2] [y2]	4
…	
[xn] [yn]	n+2
& [precision] {optional}	n+3

The parse routine starts at the second line (the first line is parsed in the global parsing driver) and builds the element until the "&" line is found. At that point, the element constructor is called and the pointer returned. An important routine used in *all* of the parse functions is the routine:

```
bool   TokenizeLine  (ifstream &BBD, char **out, int &numwords)
```

which reads the line in the input file (BBD) and returns an array of words (*out[])(character strings), as well as the number of words in the line. The function returns true if the end of the file has been reached.

WriteOutput:

The final boundary condition-level routine is the ***WriteOutput()*** routine, which writes the output associated with the element to one or more output files. Most elements write to the file "errors.txt", which contains the absolute and relative errors along the elements borders (at control points) in the form [X Y rel_err abs_err]. Additional output files may be developed at any point, such as the extraction file associated with surface water features ("extract.txt"). See chapter 3 for a discussion of some existing output files.

Listing of Boundary Condition-level element classes

A list of the currently implemented BC-level subclasses are shown in table 2.3.3a

Boundary Condition Sub-Classes

Class Name	Parent Class	Represents	Generic BC met	Special member data
CDischargeWell	CPointElem	Discharge-specified well	$Q=Q_{spec}$	--
CHeadSpecifiedWell	CPointElem	Head-specified well	$\phi=\phi_{spec}$	head,radius
CDryWell	CPointElem	Drying extraction well	$Q=Q_{spec}$ if $\phi>0$	radius
CPtDipole	CPointElem	Point Dipole	$\sigma=\sigma_{spec}$	--
CVortex	CPointElem	Point vortex	$\sigma=\sigma_{spec}$	--
CCirInhom	CCircleElem	Circular inhomogeneity in hydraulic conductivity	$\phi^+=\phi^-$	k
CCirLake	CCircleElem	Head-specified circular lake	$\phi=\phi_{spec}$	head
CRiver	CStringElem	Head specified river boundary	$\phi=\phi_{spec}$	head at vertices
CStage	CStringElem	Resistance specified rivers/streams	$\gamma(=\phi_{above}-\phi_{below})/c^*$	elevation,thickness,width,k_b,c*
CDrain	CStringElem	Drains (thin high-conductivity inhomogeneities)	$Q_t= -k/k_d b_d \Delta\Psi$	k_d, thickness
CLeaky	CStringElem	Leaky walls (thin low-conductivity inhomogeneities)	$Q_n= -k_w/kb_w \Delta\Phi$	k_w,thickness
CQSpec	CStringElem	Discharge specified rivers/surface features	$\gamma=\gamma_{spec}$	specified extraction at vertices
CQnorm	CStringElem	Normal discharge specified boundaries	$Q_n=Q_{n\ spec}$	normal discharge at vertices
CHorWell	CStringElem	Discharge-specified horizontal wells	$Q=Q_{\ spec}, \phi=const$	Q
CInhom	CStringElem	Polygonal inhomogeneities in conductivity	$\phi^+=\phi^-$	k
CBaseInhom	CStringElem	Polygonal inhomogeneities in base and/or thickness	$\phi^+=\phi^-$ or $Q_n=0$	Base,thickness
CAreaSink	CStringElem	Multi-quadric area sinks/sources	$\nabla^2\Phi=\gamma_{spec}$	leakage/recharge at points (&others)
CConductInhom	CAreaSink	Multi-quadric inhomogeneity in aquitard conductance	$\gamma=c(\phi_{above}-\phi_{below})$	conductance, pAquitard

2.4 Property Zone Classes

The property zone classes are abstractions of zones (with different geometry) within which a property value has been assigned. The property zone classes are used to represent zones of different:

- Hydraulic conductivity
- Layer Base
- Layer Thickness
- Porosity

If an element is associated with a different property (i.e. an inhomogeneity in conductivity is associated with a zone of different conductivity), it creates an instance of a property zone upon construction. These property zones are geometry independent to the end user (i.e. the Layer doesn't know if it has circular or polygonal property zones, only that it has an array of zones).

Each property zone has only two data members: its type (e.g. layer_thickness or hydraulic_conductivity) and its value (real-valued). The geometry subclasses, CCirPropZone and CPolyPropZone provide additional functionality associated with geometry (the most important being the ***GetValue***(z) routine, which returns the zone's value if the point z is inside, and the global constant *no_value* otherwise).

The static function NestedSift provides additional important functionality:

```
double NestedSift(cmplex &z, CPropZone **pZones, int nzones,
double back_val)
```

This function returns the value (i.e. conductivity) at a point z, given an array of property zones (arbitrary geometry) and a background value. The routine sifts through all of the property zones and identifies whether the point resides in any of them. If it does, then the value associated with the zone of smallest area (thus accounting for nesting of property zones) is returned. If the routine does not find a zone that contains the point, the background value is returned. This function is often used by container elements to identify point values of an aquifer attribute. The GetValue() and NestedSift() routines are greatly optimized for speed, since this routine is called so often.

2.5 Miscellaneous Classes

The primary miscellaneous classes used by the engine are CParticle, CPathline, CFlowNode, and CRectGrid

CParticle

CPathline
CFlowNode
CRectGrid

2.6 Global Drivers

The global routines operate outside of the formal object-oriented class structure, and work upon the framework described by the Bluebird library. The primary routines are:

- Parse
- Grid
- Track

Parse:

The global driver Parse takes an input file and builds the entire domain of elements and particles, assembles information on solution methods and options, identifies the user requests of the engine, and performs limited error checking on input accuracy. The input file format, which is identical to that of Split (Janković, [2000]), is read and processed to build the entire domain.

Calling syntax:

```
bool Parse(char            *filename,
           CAquifer       *&aq,
           CPathline      **particles,
           int             &numpart,
           EngineCommands  &eng)
```

Here, filename is the name of the file, aq is a pointer to the address of an empty aquifer, particles is an empty array of pointers to particles, and numpart is the number of particles. The structure eng contains (mostly boolean) information about which actions the primary driver, main(), must carry out. The EngineCommands structure is as follows:

```
struct EngineCommands{
      bool   solve;              //true if solving is on
      bool   explicitsolve;      //true if explicit solve is on
      bool   grid;               //true if gridding is on
      bool   track;              //true if tracking is on
      bool   writesol;           //true if solution should be written
      bool   writeout;           //true if output should be written
      bool   warm;               //true if solve is false
      bool   transport;          //true if transport is on
      bool   transient;          //true if transience is on
      bool   debug;              //true if debug mode on
      int    tag;                //additional debug info
      window GridWindow;         //gridding window
      int    GridResolution;     //resolution of gridding window
      double outtimes[max_times];//Transient output times
```

```
      int    nTimes;              //Number of transience output times
      double timestep;            //timestep of transient calculations
      bool   socket;              //true if a socket connection should be built
};
```

Grid:

The ***Grid()*** routine grids a set of functions (head, stream function, and leakage) at a set of points and prints them out to Golden Software Surfer™ ASCII *.grd file. These grid files may be used to then visualize contour plots. The format for these files may be found in the Surfer™ help file.

Calling syntax:

```
void Grid(int layer, const window W, int precision, const CLayer
*pLay, double t,bool debug, ofstream &PROGRESS);
```

Track:

The global driver Track, which is actually a static member function of CPathline, ***CPathline::TrackAllPathlines()*** takes an array of pathlines, a tracking duration, and tracks these pathlines through the flow domain.

Calling Syntax:

```
Void TrackAllPathlines(CPathline  **Particles,
                       const int    numpart,
                       const double timeperiod,
                       ofstream    &PROGRESS);
```

Where Particles is the array of pathlines, numpart is the array size, timeperiod is the tracking duration, and PROGRESS is an output file stream for the output of tracking progress.

3.0. Input/Output Files

There are multiple input and output files which are not independent of the object-oriented library (though some output files are created outside of the formal OO architecture).

Input files

Split.dat
The input file structure is identical to Split (Janković [2000]) and the user is referred to that format for all input with the exception of the following commands:

Dry Well
string "DryWell", string name
double x double y double Q_{pump} double r
&

Base / Thickness Inhomogeneity
string "BaseInhom", string name
double base double thickness
{double x double y}x(numlines+1)
&[int precision]

Inhomogeneity in Conductance
string "ConductInhom", string name
double conductance
{double x double y}x(numlines+1)
&[int precision]

Box of conductivity inhomogeneity cells (random k)
string "BoxOfInhomCells", double xmin, double xmax, double ymin double ymax int nx double mincond double maxcond

Vortex
string "Vortex" , string name
double x double y double strength
&

Point Dipole
string "Dipole" , string name
double x double y double strength double orientation
&

Circular Lake
string "CirLake", string name
double x double y double elev double radius
& [int precision]
Next Layer
string "NextLayer"
-used to create a new layer. All elements created after this command are assigned to this new layer.

Aquitard
string Aquitard

-used to create an aquitard beneath the current layer (as dictated by the most recent command "NextLayer". Required before adding conductance inhomogeneities of any type.

Other commands are also available, and will be added to this document as they are determined to be fully stable.

Stop
If the "stop" file (no extension) is created in the working directory, the engine stops what it is doing, writes the solution file and turns off.

Output Files

Errors.txt

An x, y, z file of errors along element borders. In the format

X	Y	rel_error	abs_error
x1	y1	e_r1	e_a1
x2	y2	e_r2	e_a2
. . .			

Extract.txt

An x, y, z file of extraction/cumulative extraction and hydraulic connection along river/lake borders. In the format:

X	Y	extraction	cum_extraction	connect
x1	y1	ex1	cex1	y/n
x2	y2	ex2	cex2	y/n
. . .				

Basemap.bna

An Atlas boundary file of element geometries. See the Surfer™ help file for format.

Superblock.bna

An Atlas boundary file of superblock geometries. Only the superblocks with elements contained are shown. See the Surfer™ help file for format.

Solution.bbs

The solution file, which stores the solved coefficients of all elements associated with the input file "split.dat". Format changes dependent upon element type & geometr, but the general format is:

Element name
* c1 c2 c3 c4 c5…cN

where c1-cN are the coefficients of the element.

Code.out, Progress.out, Debug.out

Same as in Split (Janković, [2000]). See manual for details.

4.0 Modifying/Understanding Bluebird

4.1 Adding New Element Boundary Condition Classes

The procedure for adding new element boundary condition classes is relatively simple. Just follow the steps below. As an example, we will add the useless (though realistic) QTangential class, where the tangential flux is specified along a polylinear boundary.

1) ***Create new header and source files***.

The first step to creating a new element is creating a header (*.h) file and source (*.cpp) file for the element. For the new CQTang class, we will create the new files "QTangential.h" and "QTangential.cpp" in the workspace directory. Then, in Microsoft Visual C++, we will add these to the project workspace by using the "Add files to folder" options in the "FileView" window. Initially these headers will be completely empty.

2) ***Give the header a body***

The next step to developing a new element is to create a header file for the CQTang class. The best way to do this is to use a similar class header as a template. The header for our new class will look something like this:

```
//QTangential.h

#define QTANG_H
#include "String.h"
#include "BBinclude.h"
/*****************************************************
 *  Class CQTang
 *  Analytic Tangential Discharge Specified dipole Element Data Abstraction
 *  parents: CAnalyticElem, CStringElem
 *****************************************************/

class CQSpec : public CStringElem {
 private:
   double  *Qt;     //array of tangential discharge at endpoints
   double **Qtctrl; //array of tangential discharges at all control points

 public:
  //Constructor
  CQTang(char         *Name,
         const CLayer *pLay,
         const cmplex *points,
         const double *discharge,
         const int NumOfLines);
 ~CQSpec();

  static CQSpec *Parse(ifstream &input, int &l,CLayer *pLay, char * Name);

  //Member Functions (Inherited from CAnalyticElem via CStringElem):
  void    SolveItself(double &change, double &objective,const double t);
  void    WriteOutput() const;
};
```

This header reflects the “bare bones” body of a BC-derived subclass. There is no geometric data here, and no extra accessories. The only data members are the arrays pQt and Qtctrl, which are required in order to define the boundary conditions of the element. Notice the inclusion of “String.h” and “bbinclude.h”. They are required for the class to compile and operate correctly.

3) ***Fill out the member functions***

The next step is actually writing the code. For an element whose geometric class already exists, you will only have to write the following functions in the file “QTangential.cpp”:

- Constructor
- Destructor
- Parse
- SolveItself
- GetMatrixBuildInfo (optional)
- WriteOutput

Remember to include the line

```
#include "QTangential.h"
```

At the top of the source code file (QTangential.cpp). Once again, using a similar element’s source code as a template is useful. Make sure that any dynamic arrays you create are destroyed either in the destructor (if they are data members of the class) or in the local routine in which they are created.

4) ***Clean up***

To insure that the element may be accessed by the global drivers and read from the input file you must perform the following two tasks:

a) Add the header to the global driver include file, “Bluebird.h”

For our example, this means placing the line

```
#include "QTangential.h"
```

at the top of the file “Bluebird.h”

b) Add the parse routine and parse code to the Parser in “Parse.cpp”

For our example, we first choose an unused element code (code=25) for the parse command “Qtang” (assuming this is the first word in the parsing command. We then include this line in the giant if statement at the start of the Parse Routine:

```
     ...

else if ((!strcmp(s[0],"DryWell"     )) ||
         (!strcmp(s[0],"drywell"     )) )   {code=22; }

else if ((!strcmp(s[0],"Qtang"       )) ||
         (!strcmp(s[0],"Qtangential" )) )   {code=25; }

else if ((!strcmp(s[0],"ConductInhom")) ||
         (!strcmp(s[0],"conductinhom" )) )  {code=70; }

     ...
```

Now the parser will at least recognize the first line in a parsing command. To make the parser actually parse the command, we must call the CQTang::Parse() command in the giant switch statement in the file "Parse.cpp". Place the following lines at the proper location (i.e. in numerical order of the code value (25)) in this file.

```
case(25): //-------------------------------------------------------------
 {//Tangential Discharge Specified Element
  //string "Qtang", string name
  //{double x double y double Qt}x(numlines+1)
  //&
  thisname=strcpy(thisname,blank);
  for(i=1; i<Len; i++){
    thisname=strcat(thisname,s[i]);
    thisname=strcat(thisname,space);
  }
  cast=CQTang::Parse(BBD,l,layer[curlevel],thisname);
  if (cast!=NULL){
    if (eng.warm){eng.warm=cast->ReadItself(SOLUTION,soltype);}
    if (!elemdisabled){
      AllElems[curlevel][elemsparsed[curlevel]]=cast;
  -->elemsparsed[curlevel]++;
    }
    break;
  }
}
```

c) Additional revisions to global code

If a property zone is associated with the element or other functions must be called, then additional commands may have to be added in the parse routine. Some of these are covered in the following examples. If more difficult elements are to be added, please contact the author at jrcraig2@acsu.buffalo.edu.

Problem 1: Property zones are associated with the element.

If the element coincides with a variation of conductivity, base, or thickness, the following lines must be added at the arrow in the preceding code statement:

```
PropZones  [curlevel][numpzones  [curlevel]]=tmp->GetZone();
numpzones  [curlevel]++;
```

Where GetZone() is a member function of the new class that returns the associated property zone. This function must be added as a public member of the CQTang class (CInhom is a good example of the code behind this).

Problem 2: Static class functions must be called before the element may work

Sometimes general matrices or static member data must be initialized before the element may actually operate. These functions should be placed at the bottom of the Parse routine in the "Create and Build System" section. For example, the function call

```
CInhom::SiftThroughInhoms();
```

Is called after all of the aquifer has been built. This routine sifts through all of the polygonal inhomogeneity elements looking for repeated sides, which it then disables.

If your code is in working shape, you should be able to insert an element into the input file and watch it solve itself. The best way in which to test new elements is to insure the parsing code works first, then work out the bugs in the ***SolveItself ()*** and ***GetMatrixBuildInfo ()*** code. Notice that, in order to add an element, only two existing files must be revised- the global driver for parsing input files and it's include file, Bluebird.h.

4.2 Adding New Geometric Classes

Adding new geometric classes, while a little more programming intensive than adding an element, actually requires fewer revisions to the global code (because the class is still virtual—an instance of the geometry class is useless). Though the abstractions are likely to be a bit more challenging (since the element must generalize to any boundary condition), the procedure for inserting the code into the Bluebird library is quite simple. For this example, we will use the geometry of an ellipse as the basis for a new geometry subclass.

1) ***Create new header and source files***.
The first step to creating a new geometric element class is creating a header (*.h) file and source (*.cpp) file for the element. For the new CEllipseElem class, we will create the new files "EllipseElem.h" and "EllipseElem.cpp" in the workspace directory. Then, in Microsoft Visual C++, we will add these to the project workspace by using the "Add files to folder" options in the "FileView" window. Initially these headers will be completely empty.

2) ***Give the header a body***

The next step to developing a new element geometry is to create a header file for the CEllipseElem class. Since geometric classes are so different, using another as a template may not be helpful. However, there are a few things which belong in all geometric classes:

- Information which fully describes the geometry (in global coordinates)
- Information which partially describes the geometry (in local coordinates)
- Coefficient information
- A "type flag" which indicates what type of functional structure (e.g. dipole/linesink/doublet for string elements, well/vortex/dipole for point elements)

This will most likely comprise the complete set of data members for the element. The private functions likely depend upon the implementation. For an ellipse, like a circular element, we may see no private functions, only public functions inherited from CAnalyticElem and simple accessor functions, like GetFoci(). It is important to insert the following commands at the top of the header file:

```
#include "BBinclude.h"
#include "AnalyticElem.h"
#include "Layer.h"
```

3) ***Fill out the member functions***

The next step is actually writing the code. This is much more difficult for a new element geometry, but it is likely that you've already done all of the math and coded it up elsewhere if you are about to implement it in the library. The following functions need to be filled:

- Constructor/Destructors
    - Constructor (blank)
    - Constructor (for inheriting classes)
    - Destructor
- Common but optional functions
    - SetPrecision (necessary for higher order elements)
    - Accessor Functions (e.g. GetFoci, GetRadii for an ellipse class)
- Element Functionality
    - GetDischargePotential
    - GetW
    - GetLeakage
    - GetNetDischarge
- Input/Output Functionaity
    - WriteItself
    - ReadItself
    - WriteGeometry
- Geometric Functionality
    - Centroid
    - IsInside
    - IsInSquare

    - IsInCircle
    - PartInCircle
    - GetArea
    - SharesNode
- Purely virtual (empty) functions
    - WriteOutput
    - SolveItself
    - GetMatrixBuildInfo

It is likely that most of the Geometric functionality may be found online in highly optimized c++. However, the other functions will require serious mathematical manipulation, and may require additional static matrices and functions. Once again, remember to include the line

```
#include "CEllipseElem.h"
```

At the top of the source code file (CEllipseElem.cpp). Make sure that any dynamic arrays you create are destroyed either in the destructor (if they are data members of the class) or in the local routine in which they are created.

4) ***Clean up***

To insure that the element may be accessed by the global drivers, used by element subclasses you must only include the line:

```
#include "CEllipseElem.h"
```

at the top of the file “Bluebird.h”. You may additionally need to create some global constants, such as maximum number of control points for an ellipse, or the maximum order of an elliptical element. These may be placed in “bbinclude.h”

🕮*References:*

Booch, G., ***Object-oriented analysis and design with applications***, 2nd ed, Benjamin/Cummings Publishing Co., Redwood City, CA, 1994

Haitjema, H.M., ***Analytic Element Modeling of Groundwater Flow***, Academic Press, San Diego, 394pp., 1995

Haitjema H.M. *Modeling 3-dimensional flow in confined aquifers by superposition of both two-dimensional and 3-dimensional analytic functions*, **Water Resources Research** 21 (10): 1557-1566 1985

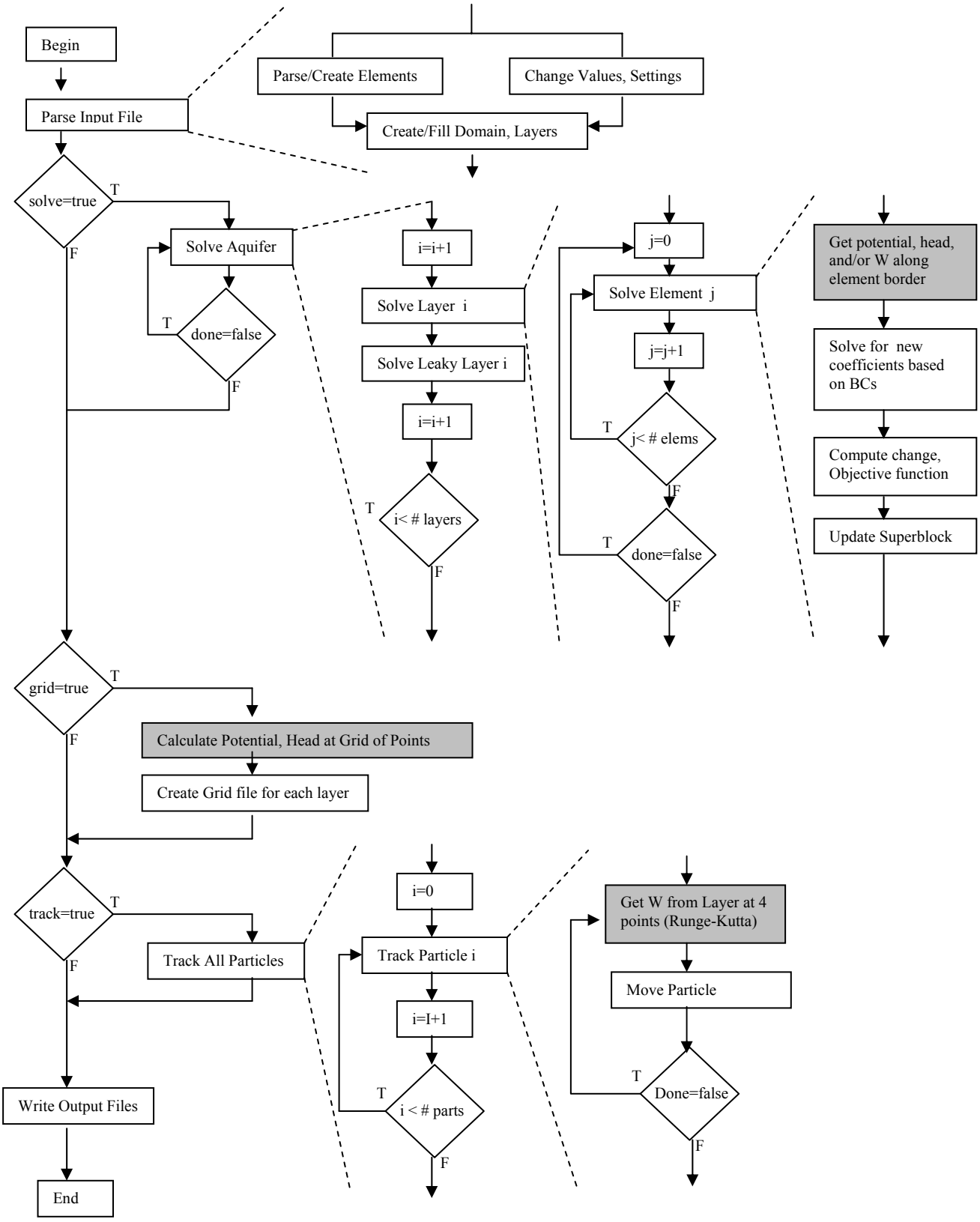
Janković, I. *High Order Analytic Elements in Modeling Groundwater Flow*, **PhD Diss, Univ. of Minnesota**, 1997

Janković, I. SPLIT: Win32 computer program for analytic-based modeling of single layer groundwater flow in heterogeneous aquifers with particle tracking, capture-zone delineation, and parameter estimation, unpublished, http://www.groundwater.buffalo.edu/software/, 2000

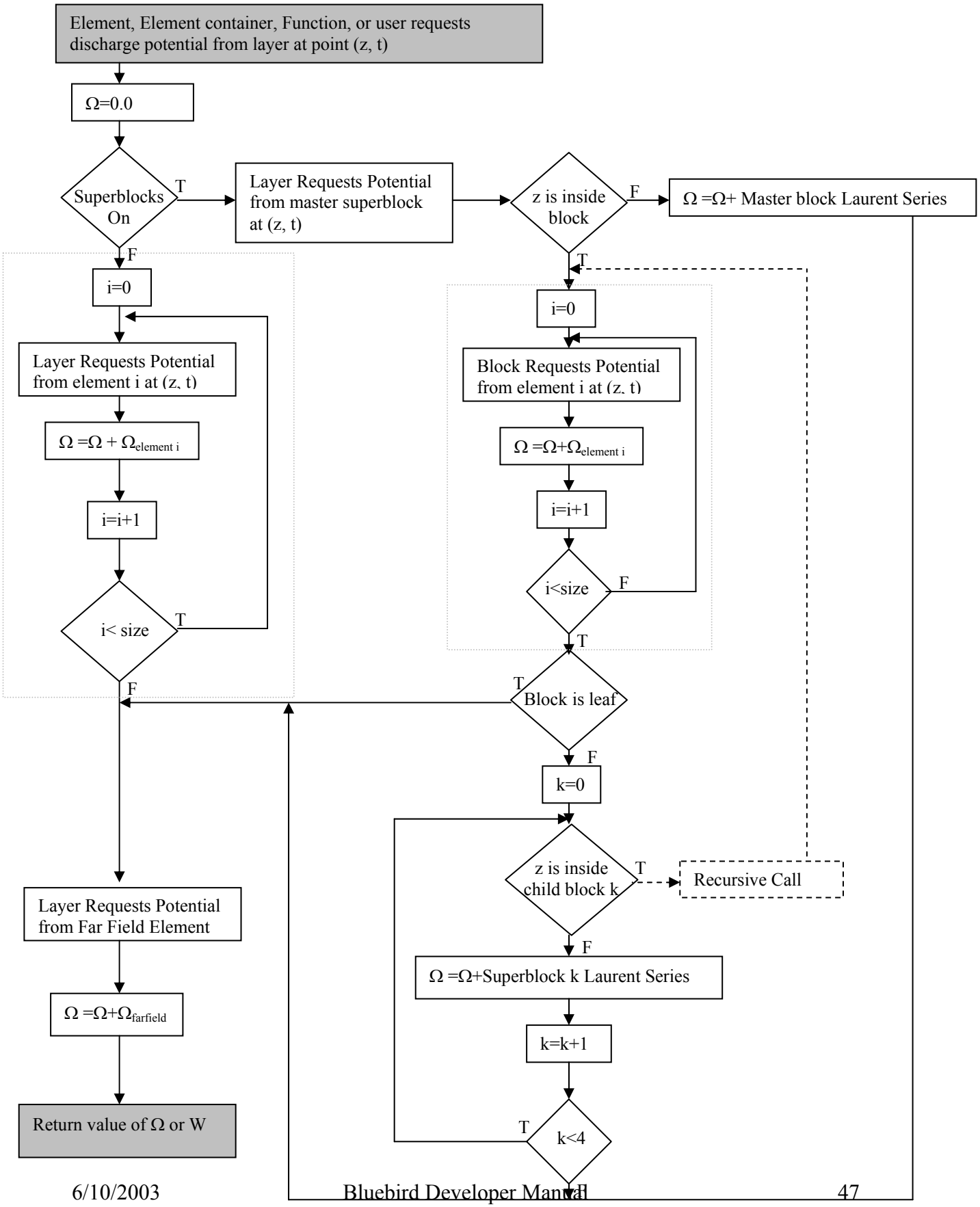
Strack, O.D.L., ***Groundwater Mechanics***, Prentice-Hall, New York, 1989

Appendix A: Bluebird Algorithm Flow Charts

Bluebird Driver Algorithm Flow Chart (Iterative Algorithm)



Bluebird Potential/W Request Flow Chart



Appendix B: Coding Conventions

In order to develop an understandable and continuous code, certain naming conventions have been maintained throughout the entire library.

Rule #1 – All Classes begin with a captial C (i.e. CAnalyticElem, CRiver, CFlowNode)

Rule #2 – Similar counting operations use the same counter variable. The variables are tabulated below:

counter	Reserved for:
i	Segments within a string Elements within an aggregate or container Columns of a matrix
j	Rows of a matrix Terms in a far field expansion
k	Inner loops (i.e. matrix multiplies) Children of a superblock
l	Lines of an input file
L	Layers within an aquifer
m	Control points along a boundary
n,n1,n2	Terms of a polynomial (i.e. Laurent series, jump function)
s	Inner loops through terms of a polynomial

Rule #3- All pointers or arrays of pointers to objects begin with the letter p. For example, each element has a pointer to the layer it is in, pLayer, and the superblock it is in, pBlock. Likewise, both layers and superblocks have an array of pointers to elements, pElemArray. This convention *does not* apply to dynamic arrays of numbers or other variables, even though they are represented by pointers in c++.

Rule #4-All variables which represent locations in the global coordinate system start with the lowercase letter z (i.e. z, zctrl, zcen, zp). Local coordinates start with a capital Z (i.e. Z, ZBH).

Rule #5- All memory is designed to be dynamic, with the exception of some temporary arrays in the global and element Parsing routines. The sizes of these temporary arrays are *not* hard coded, but are constants found in the file **bbinclude.h**

Rule #6- All input and output streams are written in all capital letters (e.g. ERRORS)

Appendix C: Index of miscellaneous functions

All of these functions are declared in **bbinclude.h**, and thus accessible by all elements and all other routines. The element bodies are located in three files: bbinclude.h, CommonFunctions.cpp, and MatrxSolvers.cpp

Simple mathematical functions

max	the maximum of two numbers (implemented for doubles & integers)
min	the minimum of two doubles (implemented for doubles & integers)
oppsign	true if the two numbers input to the function are of opposite sign
ipow	integer to a power
iabs	absolute value of an integer
random(high, low)	returns a random number between two endpoints
root	solves the quadratic equation for real roots

Zm1oZp1(cmplex Z) an optimized version of the commonly used expression:

$$X = \frac{Z+1}{Z-1}$$

Outside	calculates a complex Laurent series with complex coefficients
OutsideRe	calculates a complex Laurent series with real coefficients
OutsideW	calculates the derivative of a complex Laurent series with complex coefficients with respect to z
OutsideWRe	calculates the derivative of a complex Laurent series with real coefficients with respect to z
Inside	calculates a complex power series with complex coefficients
InsideW	calculates the derivative of a complex power series with complex coefficients with respect to z

Simple String manipulation functions

s_to_i	converts a character string to an integer
s_to_d	converts a character string to an double precision floating point number
s_to_c	converts two character strings to a double precision complex number
Countwords	counts the number of words in a space delimited character string
TokenizeLine	breaks a space-delimited character string obtained from an input file up into an array of words. Returns the array and number of words

Geometric Functions

Insquare	returns true if a given point is within a square oriented with the global coordinate system

Girinskii Potential

IsConfined	Returns true if the layer is confined at a point with provided potential, thickness, and conductivity
ConvertToHead	Converts potential to head
ConvertToPotential	Converts head to potential
GetTransmissivity	Calculates transmissivity

Global Important functions

GlobalDelete	Deletes all global dynamic arrays (the important ones).
ExitGracefully	Exits with a statement printed to the screen and the file code.out. Calls Global Delete. ***Should be put at tests in the code that otherwise might cause a run-time error.***
SetPrecision	essentially a hash table routine- enter the precision level and the type of element and get out the order and over specification fold

Matrix Manipulation Functions/Solvers

MatMult	Multiplies a Matrix (or its transpose) times a vector
Gauss	Solves the problem Ax=b using gauss elimination
PCG	Solves the problem Ax=b using a specified type of descent algorithm (i.e. conjugate gradient method)
ThomasAlgorithm	Solves the problem Ax=b (where A is tridiagonal matrix) using the thomas algorithm

Appendix D: Glossary of Terms