USGS
science for a changing world

Prepared in cooperation with the U.S. Geological Survey Water Availability and Use Science Program

Documentation for the MODFLOW 6 Framework

Chapter 57 of
Section A, Groundwater
Book 6, Modeling Techniques

Techniques and Methods 6–A57

U.S. Department of the Interior
U.S. Geological Survey

Cover. Binary computer code.

Documentation for the MODFLOW 6 Framework

By Joseph D. Hughes, Christian D. Langevin, and Edward R. Banta

Chapter 57 of
Section A, Groundwater
Book 6, Modeling Techniques

Prepared in cooperation with the U.S. Geological Survey Water Availability and Use Science Program

Techniques and Methods 6–A57

U.S. Department of the Interior
U.S. Geological Survey

U.S. Department of the Interior
RYAN K. ZINKE, Secretary

U.S. Geological Survey
William H. Werkheiser, Acting Director

U.S. Geological Survey, Reston, Virginia: 2017

For more information on the USGS—the Federal source for science about the Earth, its natural and living resources, natural hazards, and the environment—visit https://www.usgs.gov or call 1–888–ASK–USGS.

For an overview of USGS information products, including maps, imagery, and publications, visit https://store.usgs.gov/.

Any use of trade, product, or firm names in this publication is for descriptive purposes only and does not imply endorsement by the U.S. Government.

Although this information product, for the most part, is in the public domain, it also may contain copyrighted materials as noted in the text. Permission to reproduce copyrighted items must be secured from the copyright owner.

Suggested citation:
Hughes, J.D., Langevin, C.D., and Banta, E.R., 2017, Documentation for the MODFLOW 6 framework: U.S. Geological Survey Techniques and Methods, book 6, chap. A57, 42 p., https://doi.org/10.3133/tm6A57.

ISSN 2328-7055 (online)

Preface

The report describes the framework for the U.S. Geological Survey modular hydrologic simulation program, called MODFLOW 6. The program can be downloaded from the U.S. Geological Survey for free. The performance of the framework has been tested in a variety of applications. Future applications, however, might reveal errors that were not detected in the test simulations. Users are requested to send notification of any errors found in this model documentation report or in the model program to the MODFLOW contact listed on the Web page. Updates might be made to both the report and to the model program. Users can check for updates on the MODFLOW Web page (https://doi.org/10.5066/F76Q1VQV).

Acknowledgments

The U.S. Geological Survey Water Availability and Use Science Program provided financial support for the work documented herein. The authors are grateful for the constructive reviews provided by Steffen Mehl of California State University, and Daniel Feinstein of the U.S. Geological Survey. The authors are also grateful for technical input provided by Sorab Panday, GSI Environmental, and Richard Niswonger, Alden Provost, Arlen Harbaugh, Dave Pollock, and Richard Winston of the U.S. Geological Survey.

Contents

Abstract 1
Introduction 2
Concepts and Terminology 2
Primary Framework Components 4
Simulation 5
The Main Program 5
Simulation Name File 8
Timing Module 10
Solutions 11
Numerical Solution 12
Backtracking 16
Pseudo-Transient Continuation 16
Under-Relaxation Methods 17
Newton Under-Relaxation 19
Solution of the Linearized Matrix Equations 19
Models 22
Numerical Models 22
Packages 25
Exchanges 25
Numerical Exchanges 25
Utilities 27
Time Series 27
Observations 28
Memory Management 30
References Cited R–1

Figures

1. Schematic diagram showing MODFLOW 6 components. 5
2. Flowchart of the main program for the MODFLOW 6 framework 7
3. Detailed flowchart of the main program for the MODFLOW 6 framework. 9
4. Schematic diagram of MODFLOW 6 objects and their relations with other objects for a conceptual example of a parent groundwater flow model with two nested child models 10
5. Schematic diagram showing the division of simulation time into stress periods and time steps 10
6. Schematic diagram showing the `BaseSolutionType` methods called from the main program 13
7. Schematic diagram showing an example structure of the $\mathbf{A}$ matrix for three numerical models and the numerical exchanges that connect the models 14

8. Flowchart of nonlinear Numerical Solution methods called from the Calculate Procedure 15
9. Graphs showing examples of Newton under-relaxation 20
10. Flowchart of linear solution methods called from the Numerical Solution Calculate Procedure ... 21
11. Schematic diagram showing `BaseModelType` methods called from the main program 23
12. Schematic diagram showing `NumericalModelType` methods called from the main program 24
13. Schematic diagram showing `NumericalExchangeType` methods called from the main program 26
14. Graphs showing six ways in which a time series can be interpreted 29

Blank page

Documentation for the MODFLOW 6 Framework

By Joseph D. Hughes, Christian D. Langevin, and Edward R. Banta

Abstract

MODFLOW is a popular open-source groundwater flow model distributed by the U.S. Geological Survey. Growing interest in surface and groundwater interactions, local refinement with nested and unstructured grids, karst groundwater flow, solute transport, and saltwater intrusion, has led to the development of numerous MODFLOW versions. Often times, there are incompatibilities between these different MODFLOW versions. The report describes a new MODFLOW framework called MODFLOW 6 that is designed to support multiple models and multiple types of models. The framework is written in Fortran using a modular object-oriented design. The primary framework components include the simulation (or main program), Timing Module, Solutions, Models, Exchanges, and Utilities. The first version of the framework focuses on numerical solutions, numerical models, and numerical exchanges. This focus on numerical models allows multiple numerical models to be tightly coupled at the matrix level.

Introduction

MODFLOW is a popular open-source groundwater flow model distributed by the U.S. Geological Survey. For over 30 years, the MODFLOW program has been widely used by academics, private consultants, and government scientists to accurately, reliably, and efficiently simulate groundwater flow. With time, growing interest in surface and groundwater interactions, local refinement with nested and unstructured grids, karst groundwater flow, solute transport, and saltwater intrusion, has led to the development of numerous MODFLOW versions. Although these MODFLOW versions are often based on the core MODFLOW version (presently MODFLOW-2005), there are often incompatibilities that restrict their use with other MODFLOW versions. In many cases, development of these alternative MODFLOW versions has been challenging due to the underlying program structure, which was designed for the simulation of a single groundwater flow model using a regular MODFLOW grid consisting of layers, rows, and columns.

A new object-oriented framework called MODFLOW 6 was developed to provide a platform for supporting multiple models and multiple types of models within the same simulation. In the new design, any number of numerical models can be included in a simulation. These models can be independent of one another with no interaction, they can exchange information with one another, or they can be tightly coupled at the matrix level by adding them to the same numerical solution. Transfer of information between models is isolated to exchange objects, which allow models to be developed and used independently of one another. Within this new framework, a regional-scale groundwater model may be coupled with multiple local-scale groundwater models. Or, a surface-water flow model can be coupled to multiple groundwater flow models. The framework naturally allows for extensions to include the simulation of solute transport.

This report provides an overview of the MODFLOW 6 framework; it begins with a description of the concepts and terminology upon which the object-oriented framework is constructed. The remainder of the report describes the primary framework components, which include the Simulation (or main program), Timing Module, Solutions, Models, Exchanges, and Utilities. The first version of the framework focuses on numerical solutions, numerical models, and numerical exchanges. This focus on numerical models allows multiple numerical models to be tightly coupled at the matrix level. The first model to be released in the MODFLOW 6 framework is the Groundwater Flow (GWF) Model. The GWF Model is described in a companion report by Langevin and others (2017).

Concepts and Terminology

The simulation framework is based on four major components: models, exchanges, solutions, and a timing module. A model solves a hydrologic process. For example, the GWF Model (Langevin and others, 2017) solves the groundwater flow equation using a control-volume finite-difference method. Other models that may be added to the simulation framework include flow through a linear network, surface-water flow, landscape hydrologic processes, and transport, for example. A solution solves one or more hydrologic models that it contains. The Numerical Solution is the first solution developed for the framework and is described in this report. It solves a nonlinear system of equations using iterative methods. An exchange facilitates the communication between two models. For example, the GWF-GWF Exchange, which is described by Langevin and others (2017), connects two GWF Models and allows cells in one GWF Model to be hydraulically connected to cells in another GWF Model. Lastly, the timing module controls the lengths of time steps and determines when the end of the simulation has been reached.

MODFLOW 6 is written in Fortran using a modern programming style. Many of the simulation components were programmed using an object-oriented design. Adams and others (2009) provides a comprehensive description of the details for implementing object-oriented programming concepts using Fortran. The object-oriented design is different from the procedural program design that was used for previous MODFLOW versions. The move to an object-oriented design for MODFLOW was necessitated by several factors: (1) The

capability for a simulation to represent any number of models of the same or different type is straightforward to implement with an object-oriented design. (2) Objects allow for complexity to be managed and organized in compartmentalized and independent pieces of code. In many cases, programmers need functionality, which can be provided by an object. Functionality can be accessed without the programmer having to understand the underlying details. (3) An object-oriented design can make it easier for others to add new capabilities. Through inheritance and the capability to override routines, new hydrologic models and model packages can be added with relative ease and without repeating lines of code contained in other parts of the program. These factors allow MODFLOW to expand and increase in complexity, as necessary, in order to simulate new hydrologic problems.

Object-oriented programming is based on four principles, which are briefly described below in the context of MODFLOW 6.

- Encapsulation—Object-oriented programming is based on the idea that data and the routines (subroutines and functions) that operate on the data are stored together as part of an object. This concept of storing data and routines together is commonly referred to as encapsulation. Data items stored within an object are called members (or attributes). Members can be intrinsic variables or objects themselves. The routines stored within an object are referred to as methods. Encapsulation often implies that some of the information within the object is hidden from other parts of the program. The protection of information in an object can prevent errors and unintended consequences. It also allows objects to be designed independently and used without other programmers having to know the details of the object.

- Abstraction—Object-oriented programs are written using classes. A class is a unit of computer code that defines the object. A class defines the members and methods that comprise the object. Classes are often described as "blueprints" for creating objects. An object is created from the class through the process of "instantiation." Instantiation means that an instance of the object has been created from the class blueprint. Multiple object instances can be created from a single class. The term "abstraction" is intended to represent the concept that the class defines the properties and behavior of an object, once it exists. A Fortran class is created using a "derived type." For example, the Fortran code that defines the GWF Model class is contained within a derived type. For a simulation involving local grid refinement, multiple instances of the GWF Model class are required. One GWF Model instance is required for the parent model and other GWF Model instances are required for each of the child models.

- Inheritance—A class can be extended to form another class using the concept of inheritance. The class that is extended is called the superclass. The new class that is created is called a subclass. Classes can be extended multiple times to create a hierarchy of class definitions. Inheritance can be used as an efficient mechanism for organizing and reusing code. In MODFLOW 6, for example, all numerical models that involve the solution of a system of equations inherit from the `NumericalModelType` class. Because the GWF Model class inherits from the `NumericalModelType` class, it automatically has all of the members and methods defined in the `NumericalModelType` class as well as any members and methods defined in the GWF Model class itself. If a method of the `NumericalModelType` class does not make sense or does not apply for the GWF Model, then the GWF Model class can "override" that method by defining its own method that takes the place of the overridden parent method. Inheritance is also beneficial for rapidly implementing new functionality in MODFLOW 6. When designing a new model, exchange, or package, for example, inheriting from a parent model type will cause the new methods to be automatically called throughout the entire framework in the correct order.

- Polymorphism—An object that can dynamically change its type during program execution is considered to be polymorphic. Polymorphic objects are used extensively in the MODFLOW 6 framework. For example, there are many instances where the program must step through a list of all the model objects

contained within a simulation and perform some type of operation on each one. As the program steps through each model, a polymorphic object is temporarily used to refer to the model. The object must be polymorphic because models can originate from different classes. For example, there are many situations in MODFLOW 6 where it is necessary to loop through all the models in the simulation and do some sort of processing, such as "read and prepare." On the first pass, the temporary object points to the first model, and the read and prepare method can be called. On the second pass, the temporary object points to the second model, and again, the read and prepare method is called. This pattern is used throughout MODFLOW 6 to loop through models, solutions, and other lists of objects.

Support for object-oriented programming with Fortran was introduced with the 2003 standard (International Standards Organization, 2004). Fortran has several different types of intrinsic variables. An intrinsic variable type can be thought of as a type that is built-in to the Fortran language. Several of the common variable types include integer, real or floating point, and character strings. Fortran also supports derived types. A derived type is a customized type designed by the programmer to store multiple variables together within a single structure. Fortran also supports the concept of type-bound procedures, which can be included as part of a derived type. These type-bound procedures have access to the information stored within the derived type. Type-bound procedures can be subroutines or functions. The Fortran language allows derived types to be extended. When a derived type is extended, the new derived type consists of the variables and type-bound procedures of the superclass type as well as whatever new variables and type-bound procedures that it defines. The availability of derived types, type-bound procedures, and the extension of derived types allows an object-oriented program to be written using Fortran. To avoid confusion with the MODFLOW 6 concept of procedure, in this report the term "routine" is used to refer to subroutines and functions, which may or may not be type-bound. The term "method" refers to type-bound subroutines and functions.

Objects are typically not used at the lowest level in the MODFLOW 6 framework. For example, it would be possible to have a different object for each cell within a model. Although there may be some benefit in designing objects at a low level, testing revealed inefficiencies in memory and performance when objects were used for low-level purposes. For this reason, objects are typically used at a higher level, such as for a package or model.

Primary Framework Components

Within the MODFLOW 6 framework, a simulation consists of a single forward run, which may include multiple models. The simulation is the highest level component and is controlled by the main program. The primary components that comprise a simulation within the MODFLOW 6 framework are shown in figure 1. These components include the timing module, solutions, models, exchanges, and utilities. A component is a general term used in this report to describe a part of the MODFLOW 6 framework. A component may be a module, object, subroutine, or collection of these used to handle a part of the program function. The components are shown in figure 1 using dashed lines to indicate that they are not object instances, but rather the modules, subroutines, and classes that define the components.

The TimingModule implemented in the present MODFLOW 6 framework is consistent with previous MODFLOW versions. The TimingModule divides the simulation period into time steps and stress periods. The TimingModule also sets a flag for the last time step of a stress period and the last time step of the simulation. Details on the TimingModule are described later in this chapter.

A solution solves one or more models and the exchanges between models. `BaseSolutionType` is the superclass from which all other solution types must inherit. The `NumericalSolutionType` is one type of solution that is available in the MODFLOW 6 framework. The downward arrow between `BaseSolutionType` and `NumericalSolutionType` is used to denote that `NumericalSolutionType` is a subclass of `BaseSolutionType`. The `NumericalSolutionType` was designed specifically to solve one

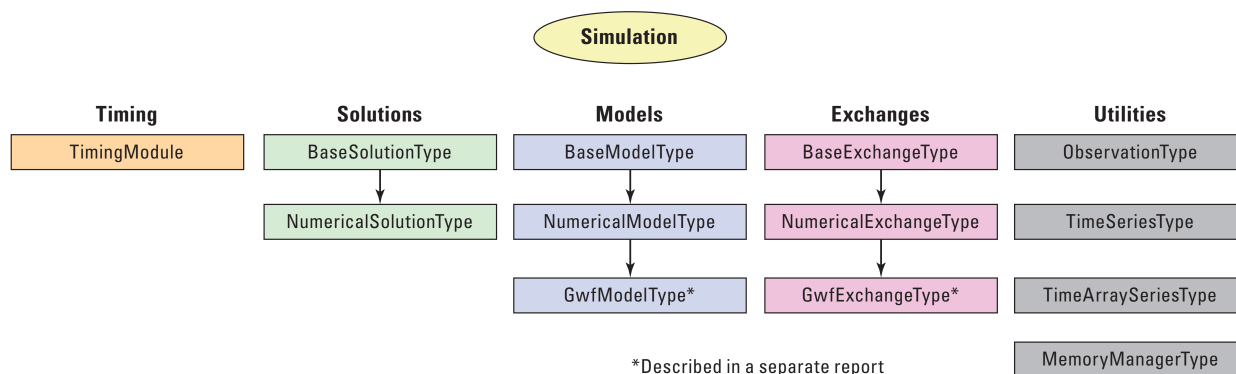


Figure 1. Schematic diagram showing MODFLOW 6 components. The `GwfModelType` and the `GwfExchangeType` are described in Langevin and others (2017).

or more numerical models, such as the GWF Model, which are subclasses of `NumericalModelType`. The `NumericalSolutionType` can also represent exchange terms between connected numerical models. These exchanges must be subclasses of `NumericalExchangeType`.

Three separate model components are shown in figure 1. Each one of these components represents a class. The arrows between these classes indicate the inheritance starting with the `BaseModelType`, extending to the `NumericalModelType`, and ending with the `GwfModelType`, which defines the GWF Model described in Langevin and others (2017). `BaseModelType` defines the members and methods shared by all models within the framework. `NumericalModelType` defines members and methods shared by all numerical models. Models that inherit from `NumericalModelType` can be solved by `NumericalSolutionType`.

Figure 1 also shows three separate exchange types. `BaseExchangeType` is the superclass for all exchanges. `NumericalExchangeType` defines exchanges between any two numerical models. The `GwfExchangeType` defines the exchange between two GWF Models. A simulation can include as many exchanges as necessary to define the problem.

Lastly, figure 1 shows several of the important utility components, which are described in the report. These include observation utility (ObservationType), the time series and time-array series utilities (`TimeSeriesType` and `TimeArraySeriesType`, respectively), and the memory manager utility (`MemoryManagerType`). The framework also includes many other minor utilities for reading and writing arrays, parsing strings, and so forth, but those minor utilities are not described in this report.

Simulation

The simulation is controlled by the main program, which is described in this section. This section also describes the simulation name file, which is prepared by the user and determines what components are active for the simulation.

The Main Program

Like all previous MODFLOW versions, MODFLOW 6 is divided into procedures, which are parts of the code that perform similar tasks. Procedures are implemented by subroutines, functions, and methods. Each procedure fulfills a well-defined purpose. The main program, which is the uppermost level of the framework,

calls procedures for the framework components in a defined order. The order of the procedure calls from the main program is shown as a flowchart in figure 2.

In a program following this flowchart, each procedure could be implemented as a single subroutine; however, the subroutines would be quite large, and there is benefit from further subdividing the work into smaller subroutines. There are many ways the code could be subdivided. The approach used for MODFLOW 6 is to divide the code into pieces that can be conceptualized and used as either procedures or components as desired. Accordingly, a primary routine is defined as the code that implements a procedure for one component. For example, solutions, models, and exchanges all have an Allocate and Read (AR) Procedure associated with them. These routines are called after Define (DF) Procedures, but before the Time Update (TU) Procedure. A procedure can be viewed as a grouping of all the primary routines that implement the procedure. A component can be viewed as a grouping of the data and all the primary routines that comprise the component. In object-oriented terms, most framework components are defined by a class that contains data and methods that implement the primary routines.

Using this approach, a simulation is simply an organized sequence of call statements to the primary routines. Each procedure is invoked through calls to multiple primary routines—one for each component. If multiple instances of a component are present, then a procedure will be called for each instance. Accordingly, the calls to the primary routines are invoked as many times as necessary based on the number of components added by the user to the simulation. Thus, the main program does not itself do all of the work of simulation, but merely calls the various primary routines in the proper sequence to do that work.

The following is a list of primary procedures that are called from the main program, as shown in figure 2.

- Create (CR) Procedure—Create framework objects, such as the model, package, exchange, and solution objects, through instantiation.
- Define (DF) Procedure—Define selected attributes for framework objects. For objects that contain allocatable arrays, the DF Procedure determines the size of these arrays so they may be allocated in a subsequent procedure.
- Allocate and Read (AR) Procedure—Allocate arrays and read information that is constant for the entire simulation.
- Time Update (TU) Procedure—Increment time variables and calculate time-step lengths.
- Read and Prepare (RP) Procedure—Read information from input files, as needed, to update hydrologic stresses or other time-varying input.
- Calculate (CA) Procedure—Update the dependent variables. For numerical solutions, the CA Procedure will use iterative numerical methods to solve the nonlinear system of equations.
- Output (OT) Procedure—Write simulation results to output files for each time step, or as required.
- Final Processing (FP) Procedure—Write termination messages and close files.
- Deallocate (DA)—Deallocate memory.

As shown in figure 2, the TU, RP, CA, and OT Procedures are called repeatedly within a time-step loop until the end of the simulation is reached.

Each box in figure 2 represents procedure calls for all models, all exchanges, and all solutions that are part of the simulation. An expanded flowchart for the simulation is shown in figure 3. This figure shows separate calls for the individual model components. For example, the AR Procedure shows calls for models, exchanges, and solutions. Each one of these boxes represents calls to all model AR Procedures, then to all exchange AR

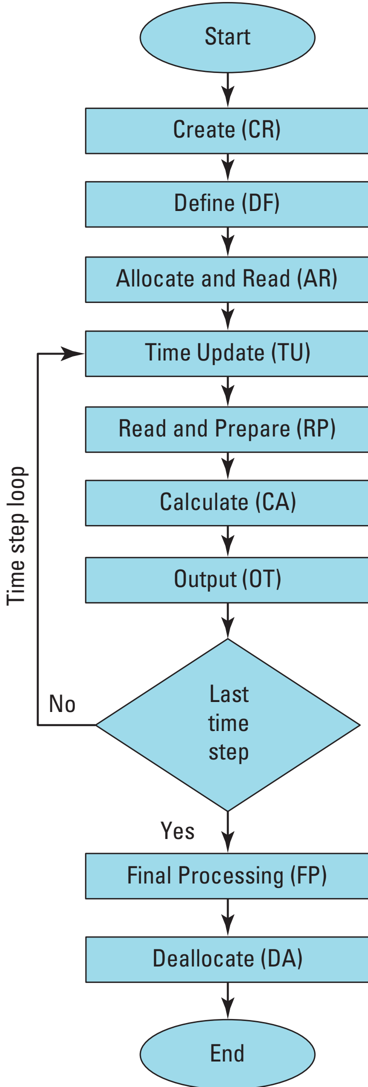


Figure 2. Flowchart of the main program for the MODFLOW 6 framework.

Procedures, and then to all Solution AR Procedures. Thus, the Model CR box shown in figure 3 represents a separate call to each Model AR Procedure. As shown in figure 3, this same pattern of calling model, exchange, and solution procedures is used for the CR, DF, AR, RP, OT, FP, and DA Procedures. The TU Procedure is called only for the Timing Module. The CA Procedure is called only for the solutions; however, as will be shown later, solutions may call model and exchange procedures as necessary in order to formulate equations.

Simulation Name File

A simulation name file is used to control the components that are active for a simulation, including the timing, models, exchanges, and solutions. Details on the format and options for the simulation name file are described in the MODFLOW 6 user guide, which is distributed with the program.

An example of a simulation name file is shown below to demonstrate the type of information that would be provided for a simulation involving three GWF Models. In this example, there is a parent GWF Model and two child GWF Models. The child models are nested within the parent model. The child models do not interact with one another. The simulation name file consists of four blocks of information that define the timing module, models, exchanges, and solutions. In this example, each block contains one or more lines of information. The first item on the line is the component type (`TDIS`, `GWF`, `GWF-GWF`, and `IMS`). The component types are also marked with a "6" in the simulation name file to indicate a version number, which may change in future releases. The second item on the line is a file name that contains information needed to define the object. Within the MODELS block, the third item on the line is the name of the model. Within the EXCHANGES block, the third and fourth items on the line are two model names. These are the models that are coupled by the exchange. Within the SOLUTION_GROUP block, the third, fourth, and fifth items on the line are the names of the models that will be solved by the Numerical Solution.

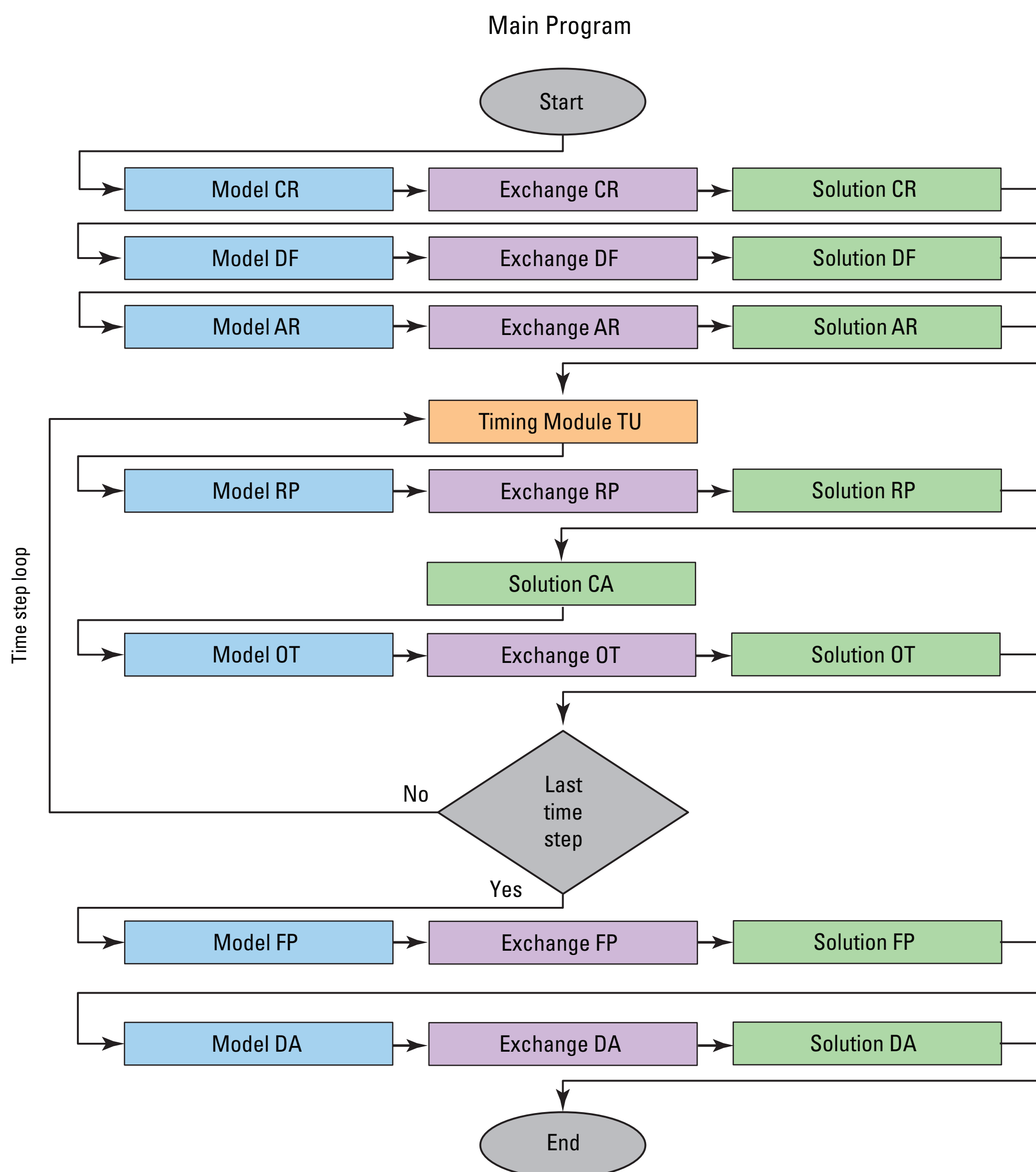
```
BEGIN TIMING
  TDIS6 simulation.tdis
END TIMING

BEGIN MODELS
  GWF6    parent.nam    PARENT
  GWF6    child1.nam    CHILD1
  GWF6    child2.nam    CHILD2
END MODELS

BEGIN EXCHANGES
  GWF6-GWF6 p-c1.exg PARENT CHILD1
  GWF6-GWF6 p-c2.exg PARENT CHILD2
END EXCHANGES

BEGIN SOLUTION_GROUP 1
  IMS6 simulation.sms PARENT CHILD1 CHILD2
END SOLUTION_GROUP
```

For the example simulation name file shown above, the MODFLOW 6 program would create the components shown in figure 4. There is the Timing Module. There is one Solution component (a Numerical Solution), which contains three models. The first model is called PARENT, the second model is called CHILD1, and the third model is called CHILD2. Exchange of information between PARENT and CHILD1 is controlled by the GWF-GWF Exchange that reads its information from the file p-c1.exg. Exchange of informa-



EXPLANATION

Procedure

CR	**Create**
DF	**Define**
AR	**Allocate and Read**
TU	**Time Update**
RP	**Read and Prepare**
CA	**Calculate**
OT	**Output**
FP	**Final Processing**
DA	**Deallocate**

Figure 3. Detailed flowchart of the main program for the MODFLOW 6 framework showing individual procedure calls for the Timing Module, Models, Exchanges, and Solutions. Each colored box in this figure may represent more than one procedure call. For example, the blue boxes represent a procedure call for each model in the simulation.

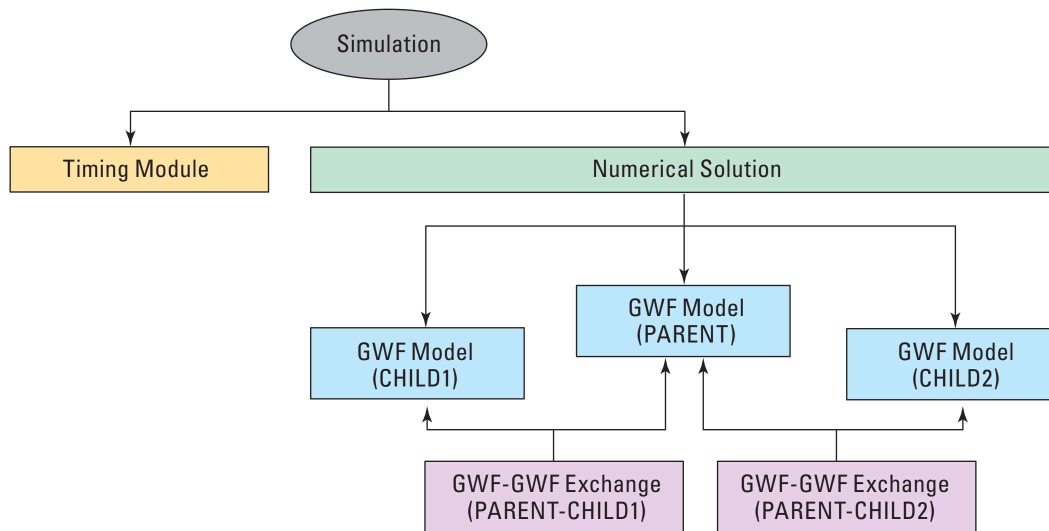


Figure 4. Schematic diagram of MODFLOW 6 objects and their relations with other objects for a conceptual example of a parent groundwater flow model with two nested child models.

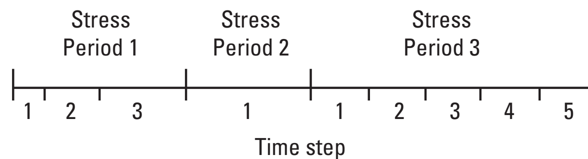


Figure 5. Schematic diagram showing the division of simulation time into stress periods and time steps.

tion between PARENT and CHILD2 is controlled by the GWF-GWF Exchange that reads its information from p-c2.exg.

For this example simulation, the three models will be tightly coupled at the matrix level and solved within a single system of equations.

Timing Module

Simulation time is divided into stress periods—time intervals during which the input data for all external stresses are constant—which are in turn, divided into time steps (fig. 5). Note that time steps are fundamental to the control-volume finite-difference method employed by the GWF Model, whereas stress periods have been incorporated in MODFLOW as a convenience for user input. Discretization information for time is read from the input file for the Timing Module (TDIS).

Within each stress period, the time steps form a geometric progression. The user specifies the length of the stress period ($PERLEN$), the number of time steps into which the stress period is to be divided ($NSTP$),

and the time step multiplier ($TSMULT$). The time step multiplier is the ratio of the length of each time step to that of the preceding time step. Using these values, the program calculates the length of the first time step (Δt_1) in the stress period as

$$\Delta t_1 = PERLEN \left(\frac{TSMULT - 1}{TSMULT^{NSTP} - 1} \right), \tag{1}$$

when TSMULT $\neq$1, and

$$\Delta t_1 = \frac{PERLEN}{NSTP}, \tag{2}$$

when TSMULT is one.

The length of each successive time step is computed as

$$\Delta t = \Delta t_{old} TSMULT. \tag{3}$$

Stress periods are implemented only as a convenience. Packages that define time-dependent stresses read input data every stress period. Stress periods facilitate the frequent need to have constant input data for stresses for multiple time steps. Situations are not unusual, however, in which a need arises to change stress data for every time step. In this situation, each stress period must consist of a single time step; alternatively, the Time Series functionality built into MODFLOW 6 can be used. The Time Series functionality is described in a separate chapter.

As an example of how stress periods are used, consider a GWF Model that uses the River and Well Packages. The simulation is for a period of 90 days, and 90 one-day time steps are used. The well and river cells are the same throughout the simulation, but the pumping rates and river stage vary. If the Time Series functionality is not used, then the number of stress periods depends on how frequently the river stage and pumping rates vary because a new stress period must start whenever stage or pumping for any cell changes. For example, if river stage or pumping only change every 30 days, then three 30-day stress periods can be used. Likewise, if pumping or river stage varies every 3 days, then 30 three-day stress periods would be used. When a new stress period begins, all stress data must be redefined; however, most stress packages will reuse the data from the previous stress period. In this example, reuse of river data would be useful if a new stress period is started because the pumping rates change while the river stage stays the same. If the Time Series functionality were used for this problem, then a single stress period could be used with 90, one-day time steps. By including the pumping rates and river stages in time-series files, the program would automatically interpolate values to the daily time step interval.

Steady-state conditions can be simulated as a single stress period with one time step. The length of a steady-state stress period does not have an impact on the computed head because the storage term in the flow equation is set to zero by the internal flow package. A time length of one unit is suggested for steady-state simulations.

Solutions

A solution is a primary component of the MODFLOW 6 framework. The purpose of a solution is to solve one or more models and the exchanges that connect them. All solutions are subclasses of

BaseSolutionType. This report documents one type of solution, the Numerical Solution, which is implemented in the NumericalSolutionType class.

The procedure calls for BaseSolutionType are shown in figure 6. These primary procedures are called directly from the main program with the corresponding procedure name. The procedures shown in figure 6 are methods of BaseSolutionType.

Numerical Solution

The Numerical Solution is designed to solve one or more numerical models that inherit from NumericalModelType. The numerical models may be connected to one another using exchanges. For these exchanges to couple models at the matrix level, the exchange must be a subclass of NumericalExchangeType. Numerical models, such as the GWF Model, formulate a nonlinear system of equations of the form

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \tag{4}$$

where $\mathbf{A}$ is the coefficient matrix, $\mathbf{x}$ is the vector of dependent variables (for example, head, stage, and concentration), and $\mathbf{b}$ is the right-hand-side vector. The Numerical Solution formulates a single system of equations (eq. 4) for all of the models and exchanges that are part of the solution. To achieve this, the Numerical Solution stores a list of the models and a list of the exchanges that it solves. To determine the structure of the coefficient matrix or to fill the matrix with coefficients, the Numerical Solution makes calls to each model in the model list and to each exchange in the exchange list. The models and the exchanges interact with the Numerical Solution by providing connection information and inserting coefficients into the coefficient matrix in the correct locations.

These concepts are illustrated in figure 7, which shows the structure of the $\mathbf{A}$ coefficient matrix for three models solved by a single Numerical Solution. This matrix is square, with the number of rows and columns equal to the total number of dependent variables in the three models. The small squares indicate a connection between two cells (n and m). A solid black square indicates that the cells are part of the same model; purple-filled squares indicate that the connection is between cells in different models.

The Numerical Solution solves the system of equations for one or more models using iterative numerical methods. The methods used by the Numerical Solution are based on the methods implemented for MODFLOW-NWT and the methods implemented in the Sparse Matrix Solver in MODFLOW-USG. The methods are capable of handling unstructured $\mathbf{A}$ matrices as well as asymmetric $\mathbf{A}$ matrices. This flexibility allows for the solution of unstructured grid problems, Newton-Raphson flow formulations, anisotropic groundwater flow, and dispersive solute transport, for example.

This system of equations solved by the Numerical Solution may be nonlinear in that the coefficient matrix may change with different values of $\mathbf{x}$. The $\mathbf{A}$ coefficient matrix may also be unstructured and can be asymmetric. As part of the CA Procedure, the Numerical Solution formulates and solves equation 4 for the models that have been added to it and for the exchanges that connect them. Because equation 4 is nonlinear, the Numerical Solution must solve a linearized form of it repeatedly, each time with an improved estimate of $\mathbf{x}$. The solution of the linearized form uses standard linear solution methods, which are described in this section under “Linear Solution.”

Methods are available in the Numerical Solution for handling the nonlinear nature of equation 4. These methods include backtracking, pseudo-transient continuation, under-relaxation methods, and Newton Dampening (fig. 8). These methods are included to adjust the estimate of $\mathbf{x}$ returned from the linear solution used to solve equation 4. In many cases, these adjustments will improve convergence.

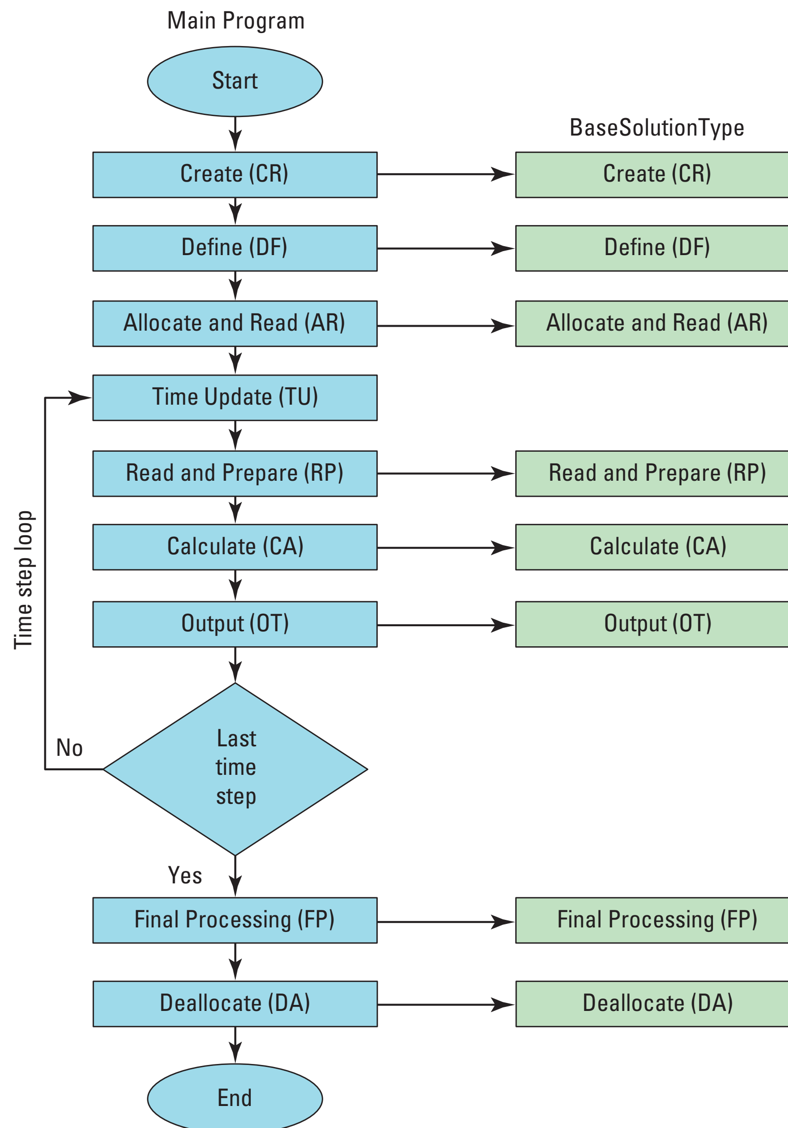


Figure 6. Schematic diagram showing the `BaseSolutionType` methods called from the main program.

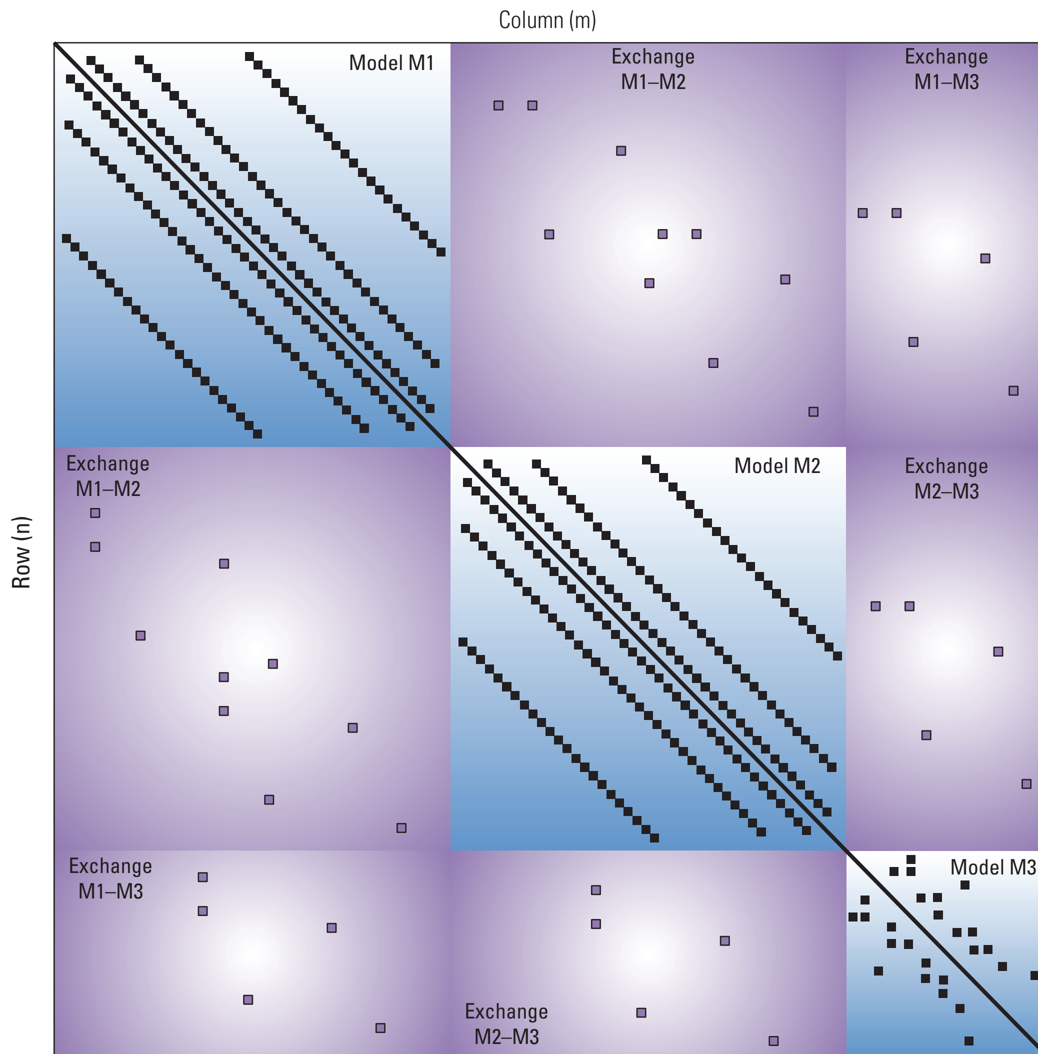


Figure 7. Schematic diagram showing an example structure of the $\mathbf{A}$ matrix for three numerical models and the numerical exchanges that connect the models.

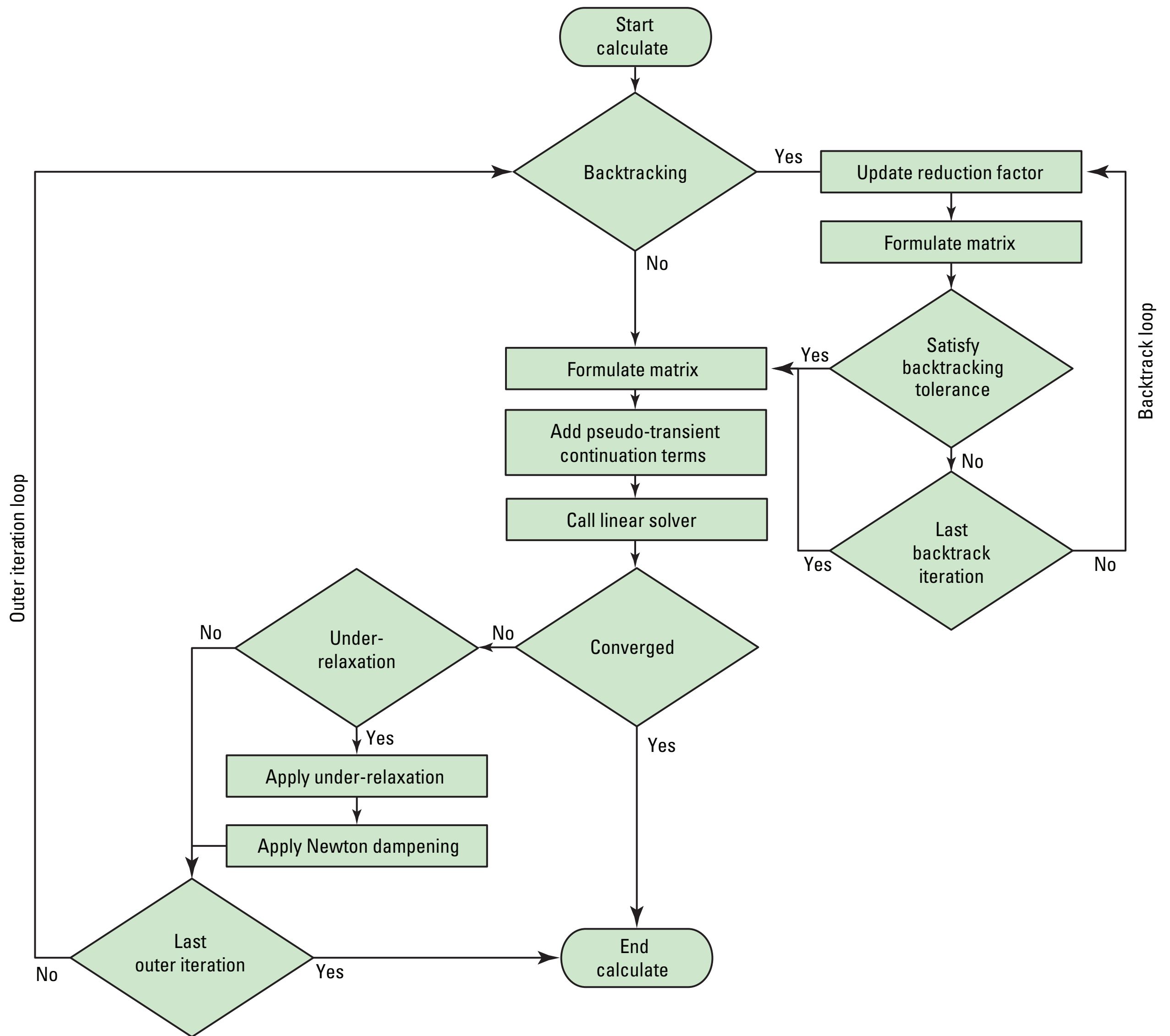


Figure 8. Flowchart of nonlinear Numerical Solution methods called from the Calculate Procedure.

Backtracking

In some cases, the Newton-Raphson method can overshoot a solution when derivatives change abruptly (for example, as a function of the dependent variable x). This condition may prevent convergence in MODFLOW 6. An option is available to use backtracking (residual control) with MODFLOW 6 if the L2-Norm of the residual increases significantly. The L2-Norm of the residual, $||r||_2$, is calculated as

$$||r||_2 = \left(\sum_{n=1}^{\text{nodes}} r_n^2 \right)^{1/2} , \tag{5}$$

where r is the residual of equation 4.

Press (2007) provides a globally convergent backtracking scheme for Newton-Raphson solutions of nonlinear equations. Backtracking reduces $\Delta \mathbf{x}$, the upgrade vector of the dependent variable for an iteration, by multiplying by a factor less than one until the error ceases to decrease. Backtracking occurs in MODFLOW 6 can occur if the user-specified variable, `BACKTRACKING_NUMBER`, is greater than zero and the residual exceeds a user-specified tolerance. If backtracking is active, $\Delta \mathbf{x}$ is reduced according to the scheme of Press (2007), as follows:

$$\begin{aligned} \text{if } ||r||_2^k &> F_B * ||r||_2^{k-1} \\ \text{while } ||r||_2^I &> F_B * ||r||_2^{k-1} \\ \text{and } ||r||_2^I &>= ||r||_{2_{min}} \\ \text{then } \Delta \mathbf{x}^I &= R_B \Delta \mathbf{x}^{I-1}, \end{aligned} \tag{6}$$

where the I subscript denotes the backtracking iteration number, F_B is a user-specified backtracking tolerance (`BACKTRACKING_TOLERANCE`), $||r||_{2_{min}}$ is a user-defined lower bound for L2-Norm of the residual that is used to terminate backtracking iterations (`BACKTRACKING_RESIDUAL_LIMIT`), and R_B is a user-specified reduction factor (`BACKTRACKING_REDUCTION_FACTOR`). Backtracking should not be used (`BACKTRACKING_NUMBER=0`) unless MODFLOW 6 is having trouble converging with under-relaxation. A backtracking iteration differs from a standard nonlinear iteration because the Newton-Raphson correction terms are not added to the coefficient matrix ($\mathbf{A}$) for models that are using the Newton-Raphson method, and the linear equations are not solved (fig. 8).

Pseudo-Transient Continuation

Steady-state problems can be difficult to solve numerically, especially when the Newton-Raphson method is used and initial conditions are not sufficiently near the roots where the residual is zero. Standard globalization strategies, such as backtracking, often stagnate at local minima, far from the correct steady-state solution. Pseudo-transient continuation is a method that improves convergence of steady-state solutions, particularly for models using the Newton-Raphson method (Kelley and Keyes, 1998). Pseudo-transient continuation methods were not implemented in previous versions of MODFLOW, but a method is available in MODFLOW 6. The pseudo-transient continuation method can be defined as

$$\left(\frac{1}{\delta^k} \mathbf{I} + \mathbf{J}(\mathbf{x}^{k-1}) \right) \mathbf{x}^k = -\mathbf{r} + \left(\frac{1}{\delta^k} \mathbf{I} + \mathbf{J}(\mathbf{x}^{k-1}) \right) \mathbf{x}^{k-1}, \tag{7}$$

where δ is a scaled pseudo-transient time-step length and $\mathbf{I}$ is the identity matrix. Equation 7 is comparable

to the Levenberg-Marquart method (Dennis and Schnabel, 1996) with $\frac{1}{\delta^k}\mathbf{I}$ being equivalent to the Levenberg-Marquart dampening parameter. The scaled pseudo-transient time-step length is calculated using

$$\delta^k = \delta^{k-1} \frac{||r||_2^{k-1}}{||r||_2^k}, \tag{8}$$

which is a switched evolution relaxation method that increases the scaled pseudo-transient time-step length in inverse proportion to the reduction in the L2-Norm of the residual (Mulder and Van Leer, 1985). The initial scaled pseudo-transient time-step length for each time step, $\delta^{k=0}$, can be user defined (PTC_DEL0) or calculated using

$$\delta^{k=0} = \frac{\sum\limits_{\substack{n=1 \\ \text{active}>0}}^{\text{nodes}} 1}{0.1||r||_2^{k=0}}, \tag{9}$$

where active is greater than 0 if cell n is not constant or inactive for the time step and $||r||_2^{k=0}$ is the L2-Norm of the residual at the beginning of the time step.

Although the pseudo-transient continuation method was developed for steady-state problems that use the Newton-Raphson method, it can also be applied to the standard conductance formulations. The equivalent linear conductance equation with pseudo-transient continuation is

$$\left(\frac{1}{\delta^k}\mathbf{I} + \mathbf{A}(\mathbf{x}^{k-1})\right)\mathbf{x}^k = \mathbf{b} + \left(\frac{1}{\delta^k}\mathbf{I}\right)\mathbf{x}^{k-1}. \tag{10}$$

Equations 7 and 10 can also be applied to transient problems, although pseudo-transient continuation dampening is generally not beneficial for these problems.

Under-Relaxation Methods

Cooley (1983) demonstrated that the Newton-Raphson method commonly requires under-relaxation to provide stable solutions. Under-relaxation methods can also be useful for improving convergence for standard formulations that do not use the Newton-Raphson method. Under-relaxation is a method for calculating the solution for the dependent variable for a particular nonlinear iteration that weights the solution from previous iterations with the present iteration. Under-relaxation can be implemented using a simple method equivalent to the under-relaxation method in the MODFLOW-2005 PCG package (Hill, 1990a), the method proposed by Cooley (1983), or a method adapted from the delta-bar-delta technique found in neural-network literature (Smith, 1993).

The under-relaxation method of Hill (1990a) dampens the solution of equation 4 using a constant relaxation factor and the change in the dependent variable during a nonlinear iteration, $\Delta\mathbf{x}$. Solution of equation 4 results in an x value for each linear equation, and $\Delta\mathbf{x}$ is calculated as

$$\Delta\mathbf{x}^k = \mathbf{x}^k - \mathbf{x}^{k-1}, \tag{11}$$

where $\mathbf{x}^k$ is the dependent variable vector for the current iteration, and $\mathbf{x}^{k-1}$ is the dependent variable vector for the previous iteration. The new dependent variable values after Hill (1990a) under-relaxation are calculated

using

$$\mathbf{x}^k = \mathbf{x}^{k-1} + \gamma \Delta \mathbf{x}^k, \tag{12}$$

where γ (UNDER_RELAXATION_GAMMA) is the user-specified relaxation factor (unitless).

The under-relaxation method of Cooley (1983) dampens the solution of equation 4 using a relaxation factor based on the maximum dependent variable change, Δx_{max}, in the current and previous nonlinear iteration. The Cooley (1983) relaxation factor, ω, is calculated using

$$\begin{aligned} \omega^k &= \frac{1}{2|s^k|} \qquad \text{if } s^k \geq -1 \\ \omega^k &= \frac{3 + s^k}{3 + |s^k|} \quad \text{if } s^k < -1, \end{aligned} \tag{13}$$

where s^k is calculated as

$$s^k = \frac{\Delta x^k_{max}}{\Delta x^{k-1}_{max} \omega^k}. \tag{14}$$

The new dependent variable values after Cooley (1983) under-relaxation are calculated using

$$\mathbf{x}^k = \mathbf{x}^{k-1} + \omega^k \Delta \mathbf{x}^k. \tag{15}$$

The delta-bar-delta methodology for under-relaxation is more robust than the Cooley under-relaxation alternative, but also uses more memory. In this scheme, an under-relaxation factor is provided to every element of the vector containing the change in the dependent variable. If there is an oscillation in the dependent variable change from the previous iteration, the under-relaxation factor for a specified cell is reduced by a user-defined amount. If the dependent variable change is in the same direction as the previous iteration, the factor for a specific cell is incremented by a user-defined amount. A momentum term is also included that adds a user-defined fraction of the previous dependent variable update to the current one. The scheme is efficient in finding the solution to problems that exhibit oscillatory behavior in the nonlinear iterations. The weighted change in dependent variable values after delta-bar-delta under-relaxation are calculated using

$$\Delta \bar{\mathbf{x}}^k = (1 - \gamma)\, \mathbf{x}^k - \gamma \mathbf{x}^{k-1}, \tag{16}$$

where $\Delta \bar{\mathbf{x}}^k$ represents the change in the dependent variable weighted. The dependent variable is then calculated from equation 16 and additional weighting, as shown below:

$$\mathbf{x}^k = \mathbf{x}^{k-1} + \mathbf{w}^k \Delta \bar{\mathbf{x}}^k + F_M \Delta \bar{\mathbf{x}}^{k-1}, \tag{17}$$

where F_M is a constant value that is used to weight solutions from previous iterations, referred to as a momentum coefficient (UNDER_RELAXATION_MOMENTUM), and $\mathbf{w}^k$ is the weighting factor. The weighting factor is calculated in one of two ways, depending on whether the solution oscillates over nonlinear iterations. If the solu-

tion oscillates, the weighting factor is calculated as

$$w_n^k = w_n^{k-1} - \theta w_n^{k-1}, \tag{18}$$

otherwise, while w_n^k is less than one, the weighting factor is calculated as

$$w_n^k = w_n^{k-1} + \kappa. \tag{19}$$

The coefficients θ ($0 < \theta < 1$) and κ ($0 < \kappa < 1$) are themselves weighting factors that are user defined (UNDER_RELAXATION_THETA and UNDER_RELAXATION_KAPPA, respectively).

Newton Under-Relaxation

Newton under-relaxation can be applied to any flow model that uses the Newton-Raphson formulation and can greatly increase the likelihood of convergence for highly nonlinear models, particularly with cells that transition from wet to dry during the simulation. The objective of Newton under-relaxation is to limit dependent-variable changes between consecutive nonlinear iterations in model cells and reduce the occurrence of dependent-variable values below the bottom-most elevation of a flow model. Newton under-relaxation is applied after solution of linear equation 4 and application of under-relaxation. When the dependent-variable in cell n is below the bottom of the model in the cells underlying cell n, the dependent variable for the current iteration is adjusted by newton under-relaxation using

$$x_n^k = (1 - \beta)x_n^{k-1} + \beta zmin_n, \tag{20}$$

where β is the weight (unitless) used to scale the current value of x, and $zmin_n$ is the elevation of the lowest model cell bottom elevation underlying cell n (L). The variable $zmin_n$ is determined using cell connectivity data to identify the cell underlying cell n with the lowest bottom elevation that can receive vertical flow from cell n. In MODFLOW 6 , β is specified to be 0.9.

Examples of Newton under-relaxation behavior are shown in figure 9 for two different cases of convergence behavior. For both cases, $zmin_n$ is 10. and the initial head is 15. In the first case, the model is well behaved and the head converges on a value slightly greater than $zmin_n$ (fig. 9*A*). For the first seven iterations, the linear solution results in heads that are lower than $zmin_n$, but Newton under-relaxation raises the heads above or equal to $zmin_n$. Ultimately, the model converges on a head value greater than $zmin_n$. In the second case, the model is not well behaved, and each linear solution results in the head dropping below $zmin_n$ (fig. 9*A*). Newton under-relaxation then raises the head up to $zmin_n$. This model also converges as subsequent heads (after Newton under-relaxation) ultimately converge on a value at or slightly above $zmin_n$.

Solution of the Linearized Matrix Equations

Solution of the nonlinear system of equations requires repeated solution of a linearized matrix equation. Solution of the linearized matrix equation uses preconditioned iterative methods for an unstructured coefficient matrix. The coefficient matrix generated by MODFLOW 6 is always stored in an unstructured format, even if the problem is structured. The UPCG solver of Hughes and White (2013) has been extended to include (1) both conjugate gradient (CG) and biconjugate gradient stabilized (BiCGSTAB) linear accelerators to solve the symmetric system of equations arising from confined flow or from unconfined flow formulated using conductance and asymmetric systems of equations arising from the Newton-Raphson formulation, perched con-

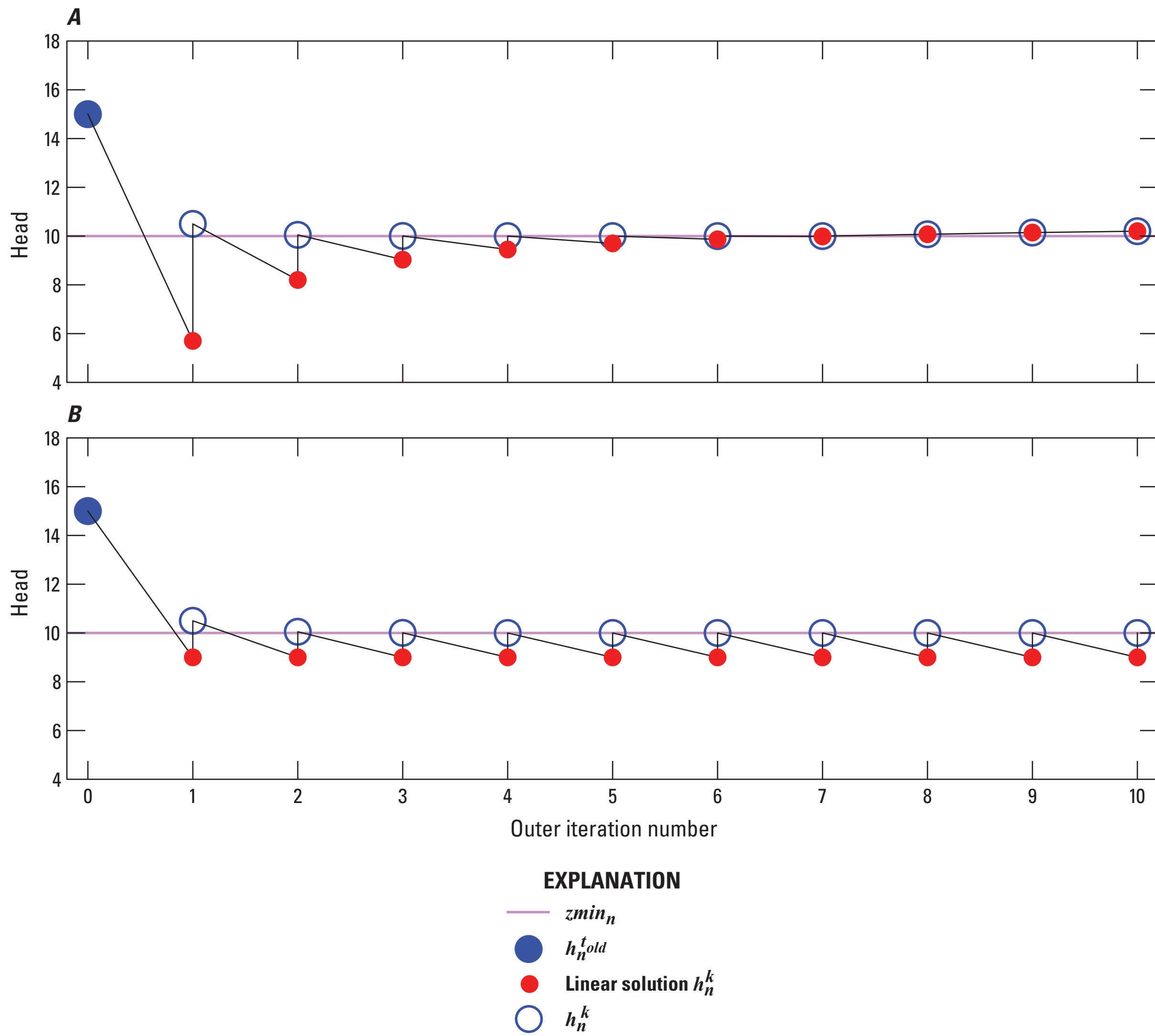


Figure 9. Graphs showing examples of Newton under-relaxation. *A*, the model is well behaved and converges to a value slightly greater than $zmin_n$. *B*, the model is not well behaved and bottom averaging is required to raise the head up to $zmin_n$.

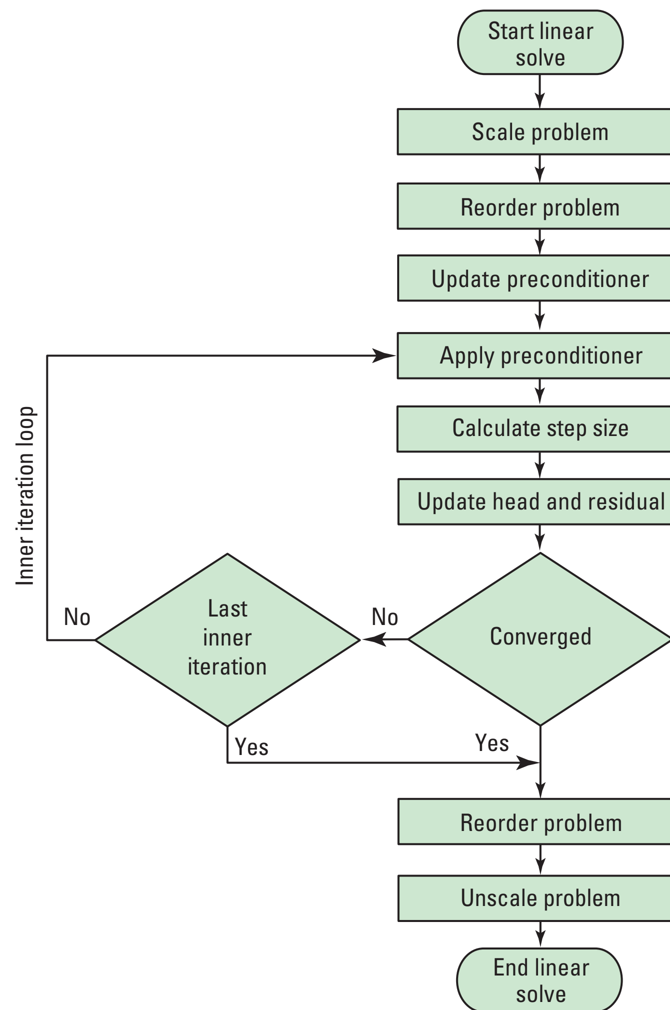


Figure 10. Flowchart of linear solution methods called from the Numerical Solution Calculate Procedure.

ditions, and ghost nodes, respectively (Barrett and others, 1994); (2) options for various levels of fill for ILU decomposition used as a preconditioning step with additional drop-tolerance schemes for additional efficiency (Saad, 1994a and Saad, 1994b) and options for row-sum agreement (Gustafsson, 1979, Ashcraft and Grimes, 1988, and Hill, 1990b); and (3) options for matrix reordering (Cuthill and McKee, 1969 and George and Liu, 1989).

The preconditioned CG and BiCGSTAB iterative linear accelerators are efficient methods for solving large systems of linear equations having a square, symmetric, positive-definite coefficient matrix and a square, asymmetric, positive-definite coefficient matrix, respectively. The flowchart for the linear portions of the solution framework is shown in figure 10. More information on the preconditioned CG and BiCGSTAB iterative linear accelerators can be found in (Barrett and others, 1994) and (Saad, 2003).

Models

A model is a primary component of the MODFLOW 6 framework. A model is intended to represent a hydrologic process, such as groundwater flow, laminar or turbulent flow in conduits, surface-water flow, solute or heat transport, or landscape hydrological processes, for example. All models should be subclasses of `BaseModelType`, as shown in figure 1.

The procedure calls from the main program to `BaseModelType` methods are shown in figure 11. These methods are empty placeholders that can be overridden by models that are subclasses of `BaseModelType`. Thus, if a new model is added that is a subclass of `BaseModelType`, then overriding methods of the new model will automatically be called by the main program, as shown in figure 11. This design makes it relatively easy to add a new model into the framework without having to make changes to the main program.

Numerical Models

The Numerical Model, defined by `NumericalModelType`, is a special type of model that is designed to work with the Numerical Solution. Consistent with previous MODFLOW versions, a numerical model can be based on individual packages to handle specific aspects of the model function. Calls to `NumericalModelType` methods are shown in figure 12. Calls to these methods are made from the main program and also from the Define and Calculate procedures of the `NumericalSolutionType`. If the model implements packages, then these model methods will likely make calls to package methods. The following is a description of the procedures of the Numerical Model that are called from the main program or from the Numerical Solution:

- Create (CR) Procedure—Create the Numerical Model and any model packages that are required by the model.
- Define (DF) Procedure—Define the Numerical Model by reading model size information.
- Add Connections (AC) Procedure—Add model connections to the Numerical Solution by reserving space within the coefficient matrix.
- Map Connections (MC) Procedure—Create an index array that maps the model connections within the Numerical Solution matrix equations. The index array is used in subsequent procedures to add terms to the matrix equations in the correct locations.
- Allocate and Read (AR) Procedure—Allocate model arrays and read model information that is constant for the entire simulation.
- Read and Prepare (RP) Procedure—Read model information from input files, as needed, to update hydrologic stresses or other time-varying input.
- Advance (AD) Procedure—Advance the model for the next time step, typically by storing the old value of model-dependent variables.
- Calculate Coefficients (CF) Procedure—Calculate or update coefficients that depend on results from the last iteration.
- Fill Coefficients (CF) Procedure—Calculate and add model terms to the Numerical Solution coefficient matrix and right-hand side vector.
- Newton-Raphson (NR) Procedure—Calculate and add Newton-Raphson terms for the model to the Numerical Solution coefficient matrix and right-hand side vector.

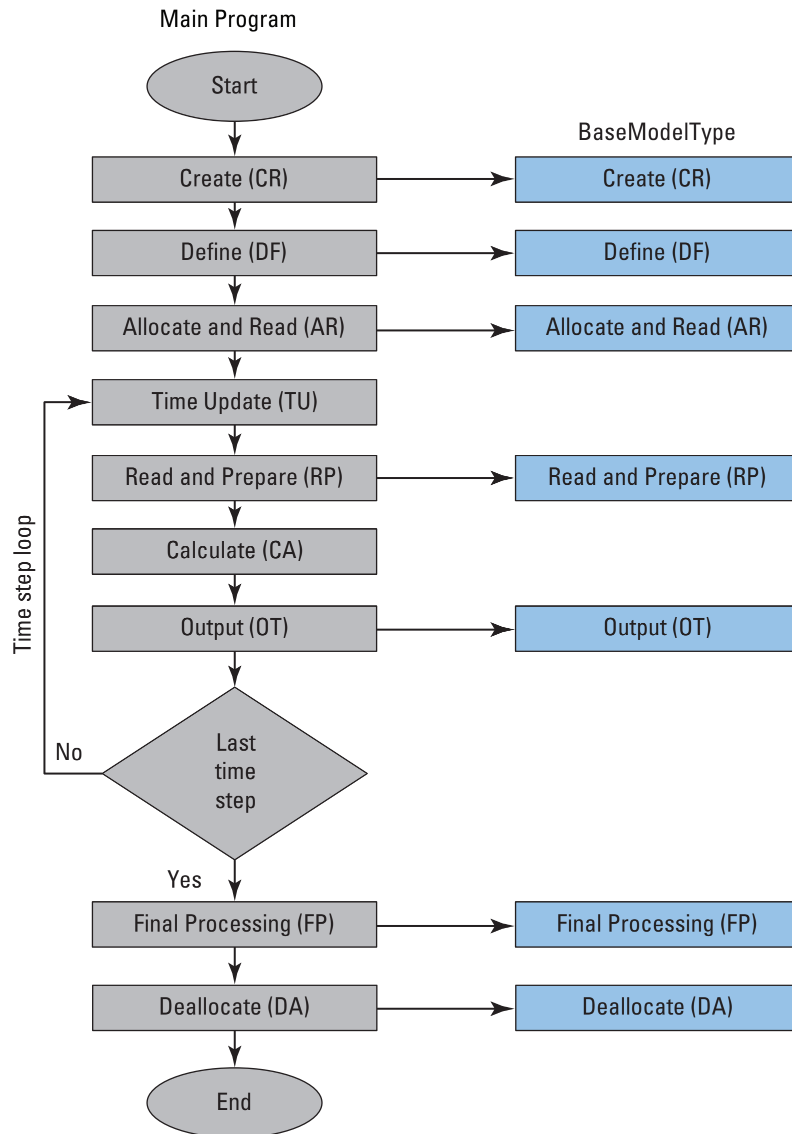


Figure 11. Schematic diagram showing `BaseModelType` methods called from the main program.

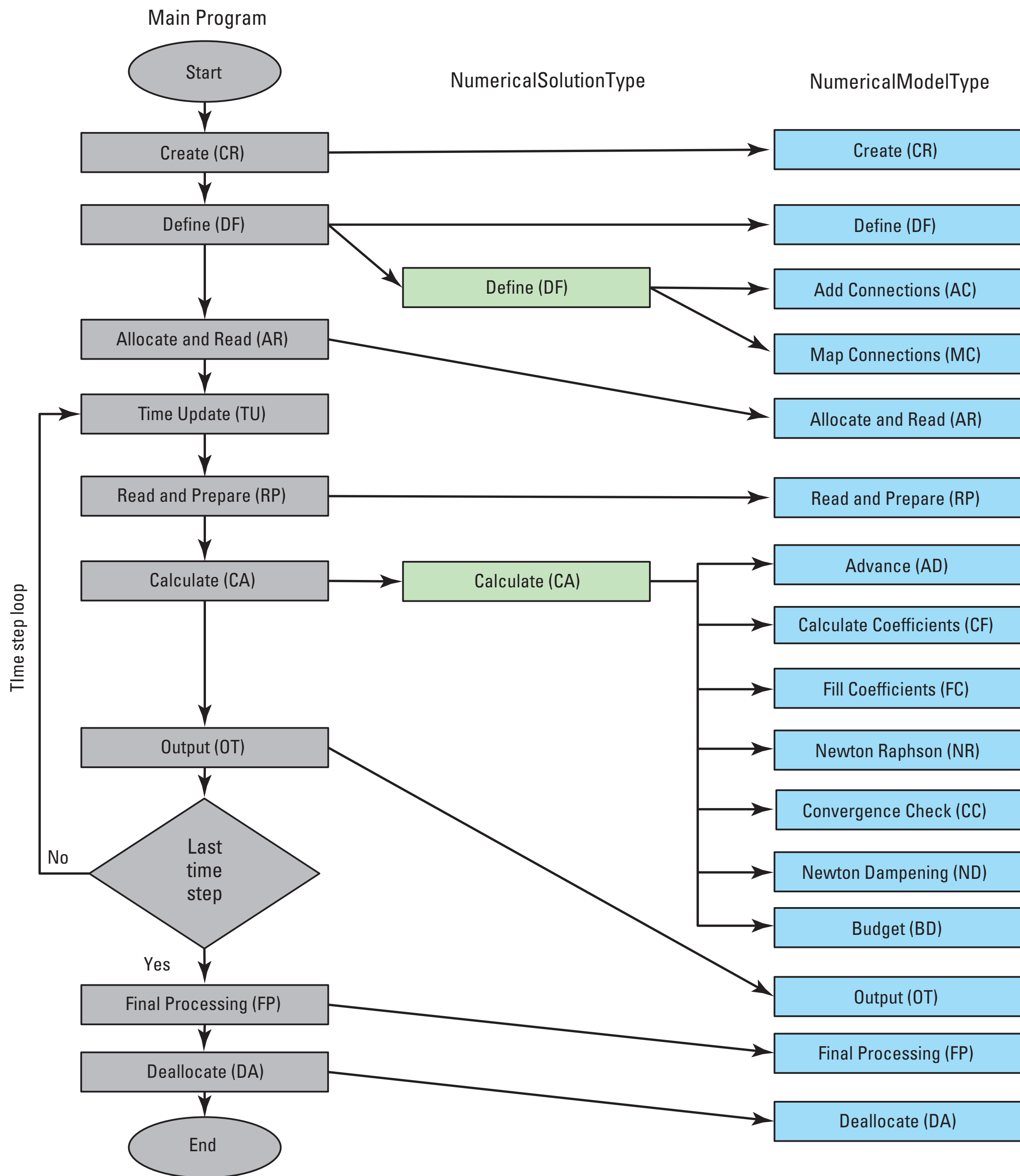


Figure 12. Schematic diagram showing `NumericalModelType` methods called from the main program.

- Convergence Check (CC) Procedure—Perform a convergence check on model-dependent variables that are not part of the Numerical Solution.
- Newton-Dampening (ND) Procedure—Adjust the calculated values for model-dependent variables. This can improve convergence for models that use a Newton-Raphson formulation.
- Budget (BD) Procedure—Calculate the model budget based on the updated solution for the dependent variable.
- Output (OT) Procedure—Write model results to output files for each time step, or as required.
- Final Processing (FP) Procedure—Write termination messages and close files associated with the model.
- Deallocate (DA)—Deallocate memory for the model.

Packages

Numerical models may be divided into "packages." A package is that part of the program that deals with a single aspect of simulation. Some boundary conditions can be implemented for a numerical model by adding to the coefficient matrix diagonal position and to the right-hand side vector. For these types of boundary conditions, the `BaseNumericalPackageType` can be subclassed to created specific boundary package types. With this implementation, the model stores a list of the boundary packages that it contains. Then as part of the Numerical Model methods, the model can iterate through the list of boundary packages and call individual package methods. This design makes it relatively straightforward to implement new types of boundary packages because the package methods will be called automatically at the correct times within the program.

Exchanges

The purpose of an exchange object is to connect or pass information between two models. The concept of an exchange was implemented as a way to maintain model independence. With the concept of an exchange, a model does not need to be updated every time there is a need to connect it to another model. Instead, only an exchange needs to be written. In order to couple two different types of models within the MODFLOW 6 framework, a new exchange class needs to be programmed. The advantage of the exchange concept is that the instructions for coupling the two models is isolated within a single exchange object. This approach makes it possible to develop models independently of other models.

Numerical Exchanges

The Numerical Exchange, defined by `NumericalExchangeType`, is a special type of exchange that is designed to work with the Numerical Solution. A Numerical Exchange can be used to add the off-diagonal terms to the A matrix, as shown by the purple squares in figure 7. Calls to `NumericalExchangeType` methods are shown in figure 13. Calls to these methods are made from the main program and also from the Define and Calculate procedures of the `NumericalSolutionType`.

The following is a description of the procedures of the Numerical Exchange that are called from the main program or from the Numerical Solution:

- Create (CR) Procedure—Create the Numerical Exchange.
- Define (DF) Procedure—Define the Numerical Exchange by reading exchange size information.

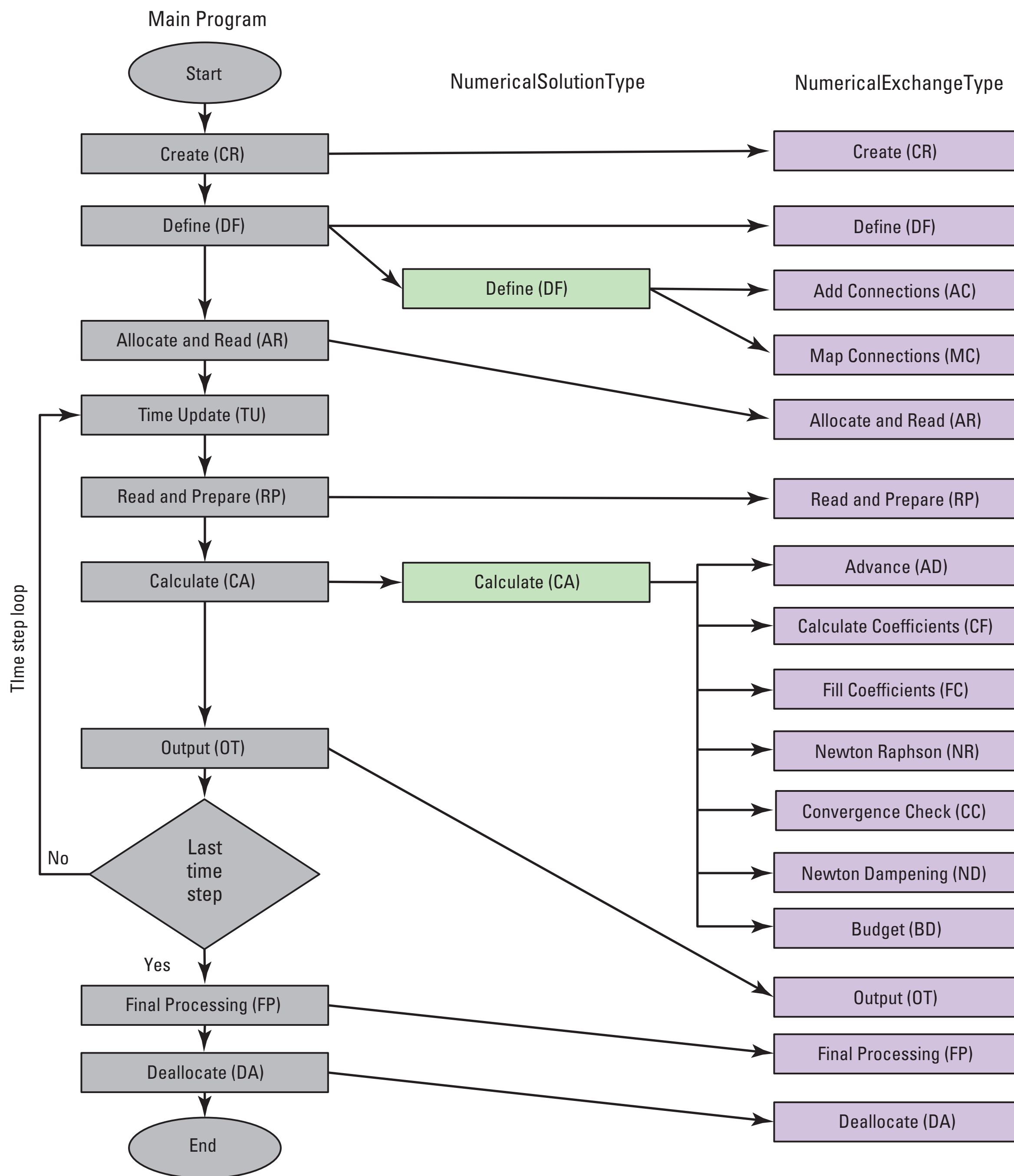


Figure 13. Schematic diagram showing `NumericalExchangeType` methods called from the main program.

- Add Connections (AC) Procedure—Add exchange connections to the Numerical Solution by reserving space within the coefficient matrix.
- Map Connections (MC) Procedure—Create an index array that maps the exchange connections within the Numerical Solution matrix equations. The index array is used in subsequent procedures to add terms to the matrix equations in the correct locations.
- Allocate and Read (AR) Procedure—Allocate exchange arrays and read exchange information that is constant for the entire simulation.
- Read and Prepare (RP) Procedure—Read exchange information from input files, as needed.
- Advance (AD) Procedure—Advance the exchange for the next time step.
- Calculate Coefficients (CF) Procedure—Calculate or update coefficients that depend on results from the last iteration.
- Fill Coefficients (CF) Procedure—Calculate and add exchange terms to the Numerical Solution coefficient matrix and right-hand side vector.
- Newton-Raphson (NR) Procedure—Calculate and add Newton-Raphson terms for the exchange to the Numerical Solution coefficient matrix and right-hand side vector.
- Convergence Check (CC) Procedure—Perform a convergence check on exchange terms.
- Budget (BD) Procedure—Calculate the exchange budget terms based on the updated solution for the dependent variable.
- Output (OT) Procedure—Write exchange results to output files for each time step, or as required.
- Final Processing (FP) Procedure—Write termination messages and close files associated with the exchange.
- Deallocate (DA)—Deallocate memory for the exchange.

Utilities

MODFLOW 6 has a variety of utility classes and routines. There are utilities for opening files, reading and writing arrays, storing information in linked lists, and so forth. There are also several larger utility functions related to time series, observations, and memory management. Descriptions for these larger utility functions are described next.

Time Series

The Time Series functionality of MODFLOW 6 allows time-dependent package input to vary from time step to time step. The input may be for such a boundary stress as well discharge rate or river stage. Alternatively, the input may be for such a boundary property as drain conductance. Time series can also be used to provide time-varying values for other package variables (such as auxiliary variables). A time-array series is similar in concept to a time series, but a time-array series is used to define time-varying input for a two-dimensional array. In effect, a time-array series can be thought of as a two-dimensional array in which each array element is a time series.

A time series is an ordered sequence of records of discrete times and corresponding values in which the time of each subsequent record is later than the time specified in the preceding record. The start of the simulation defines time zero. If a simulation contains three stress periods of length 1.0, 3.3, and 5.7 days, the start of stress period 3 would be at time 4.3 days and the end of stress period 3 would be at time 10 days. A time series used to control a stress for stress periods 2 and 3 would need, at a minimum, to encompass times from 1 day to 10 days. A time series can start before, or end after, the stress period(s) in which values are to be used. Once referenced in package input for a stress period, a time series remains in effect until the next stress period for which a PERIOD input block is listed for that package.

The time/value pairs that define a time series are provided in a file that also identifies a name and interpolation method for the time series. A time series is referenced in package input by specifying the time-series name in place of selected numeric values within a PERIOD block. When a package reads a time-series name in a position where a numeric value defining a boundary stress or other property is required, MODFLOW 6 links that time series to the numeric input. At each time step, the time series is queried to provide a value for the stress or other property. In general, a numeric value representing an average over the duration of the time step is appropriate. For example, if the boundary represents a well, one would want the time series to generate an average discharge of the well over that time step. The interpolation methods supported by time series are STEPWISE, LINEAR, and LINEAREND. The STEPWISE and LINEAR options provide a time-averaged value for a time step; the distinction between these options is the method by which values are interpolated between the time/value pairs provided in the time series. In some situations one might want the package to use a numeric value representing the time at the end of the time step, as defined by piecewise linear interpretation, rather than a value averaged over the time step; the LINEAREND option is provided to meet this need.

The effects of the three interpolation options and time discretization are shown in figure 14. In each case, the time/value pairs defined in the time series are the same (black circles). The black lines represent interpolation according to the specified interpolation method. The STEPWISE option is illustrated in figures 14*A* and 14*B*. The LINEAR option is shown in figures 14*C* and 14*D*. The LINEAREND option is shown in figures 14*E* and 14*F*. In figures 14*A*, 14*C*, and 14*E*, time steps are 1 day long. In figures 14*B*, 14*D*, and 14*F*, time steps are 2 days long. In figures 14*A* through 14*D*, the value produced by the time series for each time step is shown by the height of the blue bar. In figures 14*E* and 14*F*, the value produced by the time series for each time step is shown by the red triangle at the end of the time step.

Time-array series can be used in packages for which numeric input by arrays is supported. These packages include the Recharge and Evapotranspiration packages. The interpolation method for a time-array series may be specified as either STEPWISE or LINEAR; the LINEAREND option is not supported for time-array series.

Observations

The Observation (OBS) utility of MODFLOW 6 enables the user to specify selected model values for output to files suitable for further processing. In many cases, the model values are such model-calculated values as hydraulic head or flow rates. In other cases, the model values are properties of simulated features (for example, conductance). In contrast to earlier versions of MODFLOW, the OBS utility of MODFLOW 6 does not support specifying observed values. For consistency with earlier versions of MODFLOW, the term "observation" is retained to identify values to be extracted.

Observation output can be written to either a text or binary file. A header record containing observation names is written to the beginning of each observation file. The header is followed by one record for each time step. Each record contains the simulation time and an extracted value for each observation listed in the header. Values are written to output files as the simulation progresses. If the output file is a text file, the values can be monitored throughout the simulation.

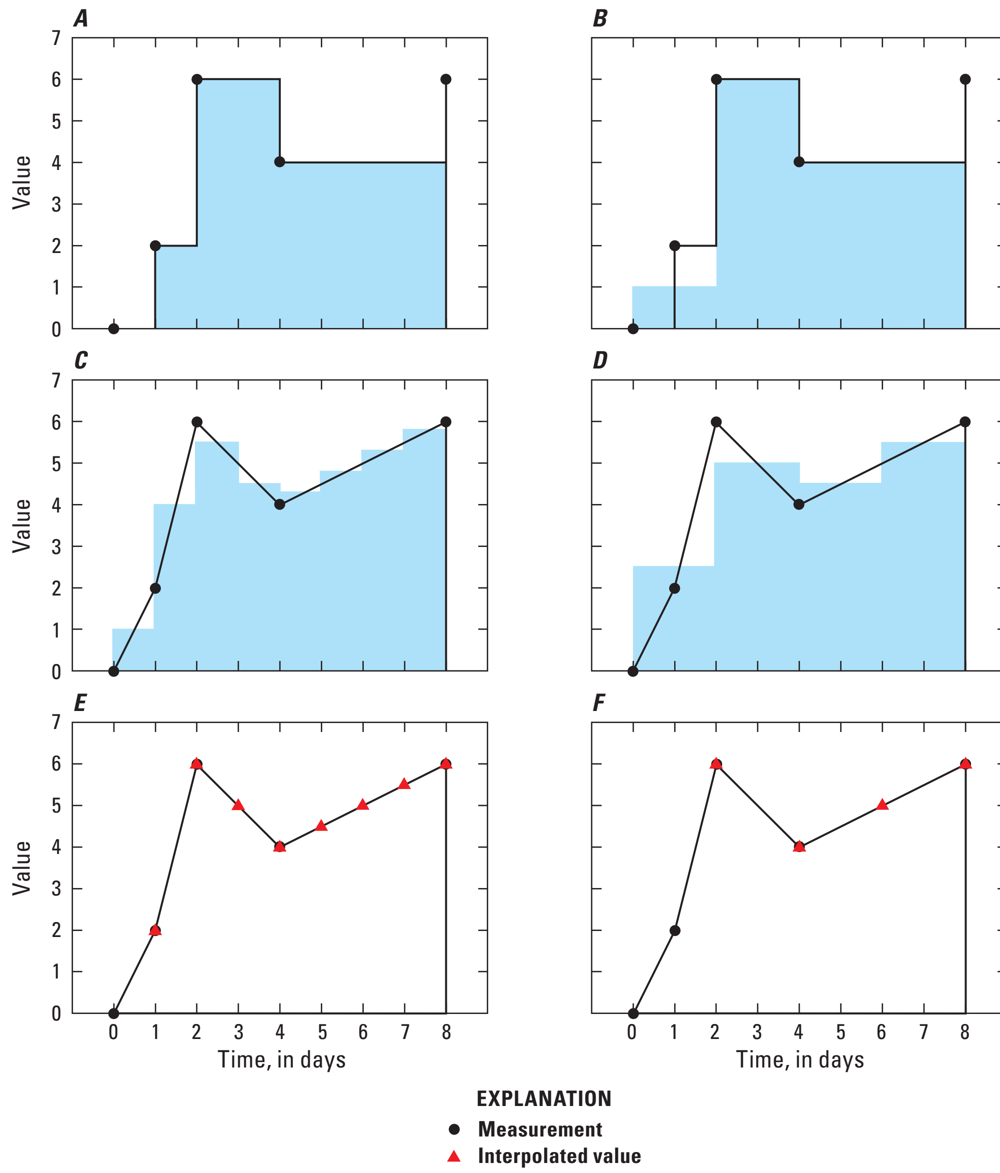


Figure 14. Graphs showing six ways in which a time series can be interpreted.

Memory Management

Most of the scalar and array variables in MODFLOW 6 are declared as Fortran pointers. Before a Fortran pointer can be used by the program, it must be allocated (or pointed to another variable). A memory manager was designed for MODFLOW 6 to serve as a centralized location for allocating these Fortran pointers. The memory manager maintains a list of all the variables in MODFLOW 6 that have been allocated using the memory-manager allocate routines. Any variable in this last can be accessed by other parts of the program using memory-manager utility subroutines. The memory manager also provides functionality for printing a table of memory usage. Routines for managing memory are located within the `MemoryManagerModule`.

References Cited

Adams, J.C., Brainerd, W.S., Hendrickson, R.A., Maine, R.E., Martin, J.T., and Smith, B.T., 2009, The Fortran 2003 handbook—The complete syntax, features and procedures, Springer Science + Business Media, 713 p., accessed June 27, 2017, at https://doi.org/10.1007/978-1-84628-746-6.

Ashcraft, C.C., and Grimes, R.G., 1988, On vectorizing incomplete factorization and SSOR preconditioners: SIAM Journal on Scientific and Statistical Computing, v. 9, no. 1, p. 122–151, accessed June 27, 2017, at https://doi.org/10.1137/0909009.

Barrett, Richard, Berry, M.W., Chan, T.F., Demmel, James, Donato, June, Dongarra, Jack, Eijkhout, Victor, Pozo, Roldan, Romine, Charles, and Van der Vorst, Henk, 1994, Templates for the solution of linear systems—Building blocks for iterative methods: Philadelphia, Penn., Society for Industrial and Applied Mathematics, 124 p., accessed June 27, 2017, at https://doi.org/10.1137/1.9781611971538.

Cooley, R.L., 1983, Some new procedures for numerical solution of variably saturated flow problems: Water Resources Research, v. 19, no. 5, p. 1271–1285, accessed June 27, 2017, at https://doi.org/10.1029/WR019i005p01271.

Cuthill, Elizabeth, and McKee, James, 1969, Reducing the bandwidth of sparse symmetric matrices, *in* Proceedings of the 1969 24th national conference, Association for Computing Machinery, p. 157–172, accessed June 27, 2017, at https://doi.org/10.1145/800195.805928.

Dennis, J.E., and Schnabel, R.B., 1996, Numerical methods for unconstrained optimization and nonlinear equations: Philadelphia, Penn., Society for Industrial and Applied Mathematics, 378 p., accessed June 27, 2017, at https://doi.org/10.1137/1.9781611971200.

George, Alan, and Liu, J. W.H., 1989, The evolution of the minimum degree ordering algorithm: Siam review, v. 31, no. 1, p. 1–19, accessed June 27, 2017, at https://doi.org/10.1137/1031001.

Gustafsson, Ivar, 1979, On modified incomplete Cholesky factorization methods for the solution of problems with mixed boundary conditions and problems with discontinuous material coneefficients: International Journal for Numerical Methods in Engineering, v. 14, no. 8, p. 1127–1140, accessed June 27, 2017, at https://doi.org/10.1002/nme.1620140803.

Hill, M.C., 1990a, Preconditioned Conjugate-Gradient 2 (PCG2), a computer program for solving groundwater flow equations: U.S. Geological Survey Water-Resources Investigations Report 90–4048, 25 p., accessed June 27, 2017, at https://pubs.usgs.gov/wri/wrir_90-4048.

Hill, M.C., 1990b, Solving groundwater flow problems by conjugate-gradient methods and the strongly implicit procedure: Water Resources Research, v. 26, no. 9, p. 1961–1969, accessed June 27, 2017, at https://doi.org/10.1029/WR026i009p01961.

Hughes, J.D., and White, J.T., 2013, Use of general purpose graphics processing units with MODFLOW: Groundwater, v. 51, no. 6, p. 833–846, accessed June 27, 2017, at https://doi.org/10.1111/gwat.12004.

International Standards Organization, 2004, Information technology–Programming languages–Fortran–Part 1; Base language: Geneva, International Standards Organization, 585 p.

Kelley, C.T., and Keyes, D.E., 1998, Convergence analysis of pseudo-transient continuation: SIAM Journal on Numerical Analysis, v. 35, no. 2, p. 508–523, accessed June 27, 2017, at https://doi.org/10.1137/S0036142996304796.

Langevin, C.D., Hughes, J.D., Provost, A.M., Banta, E.R., Niswonger, R.G., and Panday, Sorab, 2017, Documentation for the MODFLOW 6 Groundwater Flow (GWF) Model: U.S. Geological Survey Techniques and Methods, book 6, chap. A55, 197 p., accessed August 4, 2017, at https://doi.org/10.3133/tm6A55.

Mulder, W.A., and Van Leer, Bram, 1985, Experiments with implicit upwind methods for the euler equations: Journal of Computational Physics, v. 59, no. 2, p. 232–246, accessed June 27, 2017, at https://doi.org/10.1016/0021-9991(85)90144-5.

Press, W.H., 2007, Numerical recipes (3d ed.)—The art of scientific computing: New York, Cambridge University Press, 1,256 p.

Saad, Yousef, 1994a, ILUT—a dual threshold incomplete LU factorization: Numerical Linear Algebra with Applications, v. 1, no. 4, p. 387–402, accessed June 27, 2017, at https://doi.org/10.1002/nla.1680010405.

Saad, Yousef, 1994b, SPARSEKIT—A basic tool kit for sparse matrix computation (ver. 2): Minneapolis, Minn., Computer Science Department, University of Minnesota, accessed July 5, 2017, at http://www-users.cs.umn.edu/~saad/software/SPARSKIT/.

Saad, Yousef, 2003, Iterative methods for sparse linear systems: Philadelphia, Penn., Society for Industrial and Applied Mathematics, 528 p., accessed June 27, 2017, at https://doi.org/10.1137/1.9780898718003.

Smith, Murray, 1993, Neural networks for statistical modeling: New York, Van Nostrand Reinhold, 235 p.

Publishing support provided by the U.S. Geological Survey
Science Publishing Network, Reston Publishing Service Center

For information concerning this publication, please contact:

Office of Groundwater
U.S. Geological Survey
Mail Stop 411
12201 Sunrise Valley Drive
Reston, VA 20192
(703) 648–5001
https://water.usgs.gov/ogw/

USGS

Hughes and others—**Documentation for the MODFLOW 6 Framework**—Techniques and Methods 6–A57

ISSN 2328-7055 (online)
https://doi.org/10.3133/tm6A57