Integrating Module of the National Energy Modeling System: Model Documentation 2025

July 2025

Independent Statistics and Analysis
U.S. Energy Information Administration

www.eia.gov
U.S. Department of Energy
Washington, DC 20585

Table of Contents

1. Introduction 1
Scope and organization 1
2. Overview of the Structure of NEMS 3
Background 3
The Integration Module 4
Key Tasks 4
3. Global Data Structure 6
Pyfiler 9
Energy market data representation 9
Restart file 10
4. Integrating Module Solution Methodology 11
The NEMS iteration 11
Introduction 11
The Iteration Solution algorithm 11
The NEMS cycle 13
Introduction 13
The cycle solution algorithm 14
Parallel NEMS 16
Foresight approach 16
Discontinuities and convergence problems in NEMS 17
Expected value foresight 20
5. Managing NEMS runs 24
System Design 24
Queue Structure 24
Cycle.py 26
Tasks.py 26
Worker_start.bat 26
run_task.py 26
Run Monitor 27
Operation 30

The NEMS Report Writer 32
NEMS Report Writer Structure 32
Running the Report Writer 32
Preprocessor Program 33
Base Program and Table Programs 33
Postprocessor 34
RAN File Generator 34
The NEMS Validator 35

Table of Figures

Figure 1. Basic National Energy Modeling System (NEMS) structure and information flow........................ 4
Figure 2: Simplified representation of the iteration loop... 11
Figure 3. The supply curve cuts across the horizontal portion of the demand curve 18
Figure 4. The supply curve cuts across the vertical portion of the demand curve.................................... 19
Figure 5. Queue Process Flow.. 25
Figure 6. Show Jobs Process Flow.. 28
Figure 7: Report Writer system diagram 32
Figure 8: RAN File Generator diagram 34
Figure 9: Sample Validator Tests.. 35

Table of Tables

Table 1. Key Blocks in the NEMS global data structure 7
Table 2: Convergence variable weights by category 15
Table 3: Composite score to GPA........ 15
Table 4: Preprocessor input files 33

1. Introduction

The National Energy Modeling System (NEMS) is a long-term energy-economy modeling system of U.S. energy markets. The model is used to project production, imports, exports, conversion, consumption, and prices of many energy products, subject to user-defined assumptions. The assumptions encompass macroeconomic and financial factors, world energy markets, resource availability and costs, behavioral and technological choice criteria, technology characteristics, and demographics.

NEMS produces a general equilibrium solution for energy supply and demand in the U.S. energy markets on an annual basis.

EIA's Office of Energy Analysis develops and maintains NEMS to support the *Annual Energy Outlook* (AEO). EIA analysts perform policy analyses requested by decisionmakers in the White House; the U.S. Congress; offices within the U.S. Department of Energy, including program offices; and other government agencies. Users outside of EIA use NEMS for a variety of purposes.

NEMS was first used for projections presented in the *Annual Energy Outlook 1994*.

Scope and organization

Publication of this document is supported by Public Law 93-275, Federal Energy Administration Act of 1974, Section 57(B)(1) (as amended by Public Law 94-385, Energy Conservation and Production Act), which states, in part:

> *...that adequate documentation for all statistical and forecast reports prepared...is made available to the public at the time of publication of such reports.*

In particular, this report meets EIA's model documentation standard 2015-1, established under these laws.[1]

The individual components of NEMS are documented individually. Although the NEMS Integrating Module is a distinct component of NEMS, the Integrating Module is not by itself a model. Rather, it is a framework that connects the subject matter modules, and a component of the overall NEMS model. The Integrating module implements specific aspects of the overall modeling methodology that are not documented elsewhere. The documentation is organized accordingly.

Readers interested in a more comprehensive summary of NEMS should see the latest *The National Energy Modeling System: An Overview*.[2]

Chapter 3 describes the NEMS global data structure, which is used for inter-module communication, solution initialization and storage, and certain database operations.

[1] See https://www.eia.gov/about/eia_standards.php#standard2015_1.
[2] See https://www.eia.gov/outlooks/aeo/nems/documentation/index.php.

Chapter 4 provides the mathematical specification for the solution algorithm and describes the convergence techniques we use. Chapter 4 also documents other modeling functions of the Integrating Module, including generation of foresight assumptions and carbon dioxide emission policy routines.

Chapter 5 discusses the NEMS job queue and run management, which are used to manage NEMS runs in a distributed environment.

Chapter 6 discusses the NEMS Report Writer, which produces diagnostic tools and the published output from the model.

Chapter 7 discusses the NEMS validator.

2. Overview of the Structure of NEMS

Background

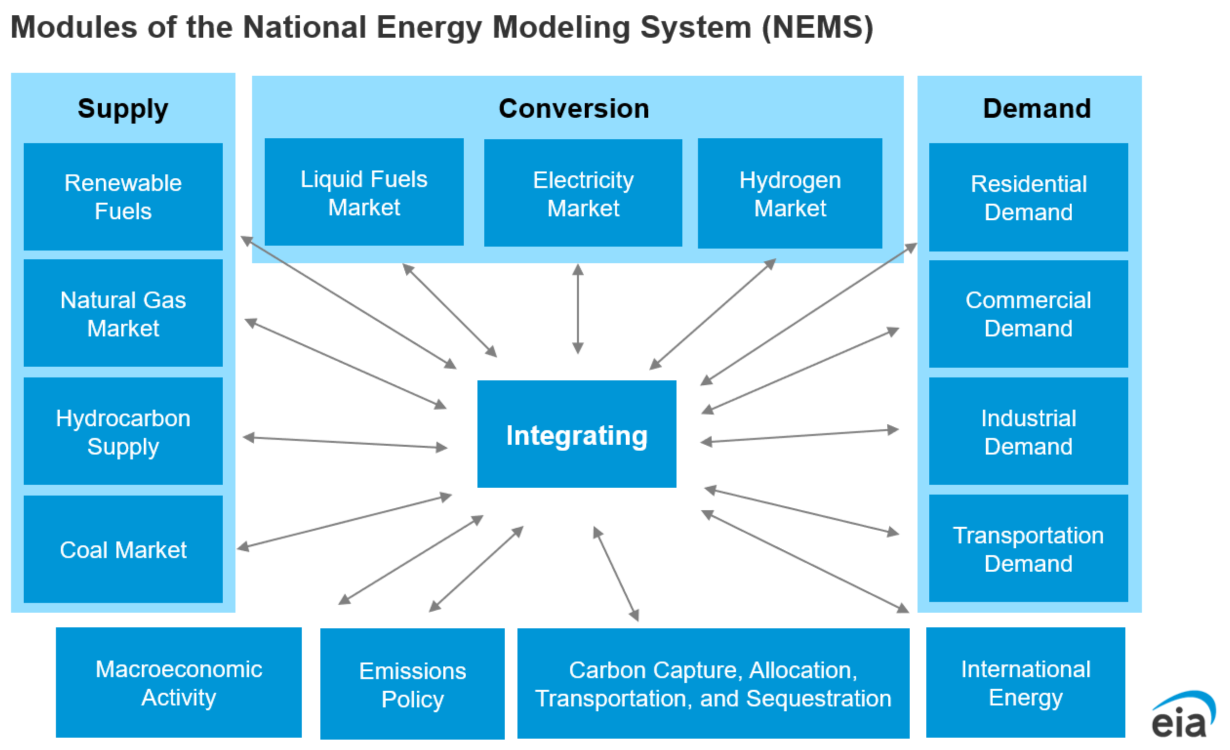
NEMS is structured as a modular system. The modules include the Integrating Module and a series of relatively independent modules that represent the domestic energy system, the international energy market, and the economy. The domestic energy system is broken down further into fuel supply markets, conversion activities, and end-use consumption sectors.

Model modularity implies a system of self-contained units, each performing a specific, well-defined function. This concept is generally consistent with the economic structure of energy markets, which can be represented by various supply, conversion, and demand components that are largely separable. Because energy markets are heterogeneous, a single methodology cannot adequately represent all aspects of supply, conversion, and end-use demand sectors. The modularity of the NEMS design provides the flexibility for each component to use the methodology and regional coverage that is most appropriate for the required analyses.

NEMS can execute the modules individually or in subsets. This flexibility fosters independent module development, a distribution of model development work organized by energy market specialties, and incremental development of the system. Several modules are further broken down into submodules for development and documentation purposes.

To support modularity, the information flow between modules is centralized. The data linkages between modules are implemented through the NEMS Global Data Structure (GDS). The Global Data Structure (discussed in more detail in Chapter 3) is the set of data communicated between the NEMS modules or used in the NEMS output reports. Individual NEMS modules access the GDS data they need for input and update the GDS variables that store their module's output.

Figure 1. Basic National Energy Modeling System (NEMS) structure and information flow



Data source: U.S. Energy Information Administration

The primary data flow among the modules are the delivered prices of energy and how much energy is consumed by product, region, and sector. The information flows among modules are not limited to prices and quantities, and they include other information such as economic activity, capital expenditures, and supply curves.

Many NEMS modules simulate the economic decision-making involved in the sector of the energy system being modeled. To represent these decisions, NEMS is constructed with reasonably fine detail of energy product categories and the regional locations of energy production and use. This detail is necessary because the economics of allocating energy products is strongly influenced by the product category at issue and regional differences in costs and other factors.

The Integration Module

Key Tasks

The integration code is the spine of NEMS. It calls most of the individual modules and manages the model's underlying functions and operations—setup, job queue, calculations, and output production.

The integration code manages operations during **setup.**

- It provides a graphical user interface and a command line interface to the system.
- It sets up the folders for a run.

- It compiles, using the meson build system, the Fortran code that is used in the run.
- It preprocesses any data that is being loaded in from exterior systems.
- It manages and loads shared configuration files.

The integration code includes the **job queue.**

- It dispatches jobs from the user, to the run queue server, and then to the worker machines.
- It activates workers (which process NEMS jobs) and manages their operations.
- It manages the RabbitMQ and celery server that dispatches the jobs.
- It provides a monitor for the job queue, to review job status.

The integration code manages **calculations** during the main NEMS loop.

- It ingests data from disk, and loads it into memory.
- It manages the flow of program calls.
- It modifies data when the modifications are cross-cutting, or the calculations are performed in the integration code for some legacy reason.
- It tests convergence, and determines when the mode should stop running. If directed, it applies a relaxation algorithm.
- The integration code writes files and reports to disk where needed.

The integration code includes the NEMS **post processes.**

- The NEMS Report Writer produces all external NEMS reports.
- The NEMS validator, a simple set of checks, evaluates whether results have errors preventing publication.
- The cleanup code manages the cleanup (deletion of temporary files, compression, etc) of NEMS files after a NEMS run completes.

3. Global Data Structure

The Global Data Structure defines the subset of NEMS variables used for communication between modules and for external reporting such as the Annual Energy Outlook Tables. The variables consist of variables shared among modules, such as prices, consumption, and macroeconomic information. The variables also include reporting variables, as well as model control parameters and assumptions.

The variables in the Global Data Structure are defined and organized in *blocks* that designate groups of variables.

The specific elements of the block structure are defined in the *include* files that contain declarations for variables. In addition, a data dictionary for the Global Data Structure includes definitions for each variable.

Table 1. Key Blocks in the NEMS global data structure

Modules filling the common block	Common block names	Description
Integrating, multiple contributors, or exogenous	QBLK	End-use sector quantities
	QMORE	Additional end-use sector quantities
	MPBLK	End-use sector prices)
	PMORE	Additional end-use sector prices
	MXQBLK	Expected quantities for foresight
	MXPBLK	Expected prices for foresight
	QSBLK	State Energy Data System historical data corresponding to QBLK
	NCNTRL	Control variables
	COGEN	Combined heat and power
	CONVFACT	Thermal conversion factors
	CONVERGE	Convergence variable data and reporting summary
	COALEMM	Variables exchanged between the Coal Market Module and the Electricity Market Module
	HMMBLK	Hydrogen module variables (future use)
	CYCLEINFO	Current cycle number and total cycles in overall run
	CONTINEW	Information related to continuation of cycling
	NCHAR	Character variables such as scenario name or module names
Emissions	EMABLK	Price adjustments for carbon dioxide fees, if any
	EMEBLK	Carbon dioxide emissions factors by fuel/sector
	EPMBANK	Parameters for an emissions constraint banking option
	REGCO2	Regional carbon dioxide emissions by fuel and sector
	GHGREP	Greenhouse gas abatement costs and offsets
	EMISSION	Emissions and related results
	AMPBLK, ANGTDM, ACOALPRC, APMORE, AEUSPRC, APONROAD	Copies of MPBLK, NGTDMOUT, COALPRC, PMORE, EUSPRC, and PONROAD with prices adjusted by any energy tax or emission allowance fees
	AB32	California Assembly Bill 32 cap and trade variables
	RGGI	Regional Greenhouse Gas Initiative variables
	CSAPR	Cross-State Air Pollution Rule variables
	EMOBLK	Emissions
	CALSHR	California shares for estimating AB32 covered emissions
	INDEPM	Cement-related CO2 process emissions passed from IDM to EPM
Macroeconomic	MACOUT	Output variables
	MCDETAIL	Reporting variables
International Energy	INTOUT	All International Energy Module global variables
Residential Demand	RESDREP	Reporting variables
	RSCON	Energy consumption by end use
	RSEFF	Energy efficiency by end use

Table 1. Common Blocks in the NEMS global data structure (continued)

Modules filling the common block	Common block names	Description
Commercial Demand	COMPARM	Control parameters, assumptions
	COMMREP	Reporting variables
	BLDGLRN	Cumulative shipments of distributed generation technologies for *learning* curves
Industrial Demand	INDOUT	Industrial variables for use in other modules
	INDREP	Industry-level consumption reporting variables
	INDREP2	Industry-level combined-heat-and-power reporting variables
	BIFURC	Energy by fuel/region classified by covered and uncovered industry groups for emission cap and trade analysis
Transportation Demand	TRANREP	All global transportation variables
Electricity Market	UEFPOUT	Electricity pricing outputs
	EFPOUT	Electricity pricing outputs
	UEFDOUT	Fuel-dispatch outputs
	UDATOUT	Electricity central data outputs
	UECPOUT	Capacity planning outputs
	DSMTFEFP	Demand side management/electricity pricing
	UETTOUT	Electricity trade outputs
	EUSPRC	Electricity prices for end uses by sector
	CAPEXP	Capital expenditures
	TCS45Q	Variables for modeling U.S. tax code section 45Q credits
	ULDSMOUT	DSM variables
	E111D	EMM/CMM interface
Carbon Capture	CCATSDAT	Carbon capture, transport and sequestration variables.
Renewable Fuels	WRENEW	All Renewable Fuel Module global variables
Hydrocarbon Supply	OGSMOUT	All Hydrocarbon Supply Module global variables
Natural Gas Market	NGTDMOUT	Output variables
	NGTDMREP	Reporting variables
	NGRPT	Supplementary reporting variables
Liquid Fuels Market	PMMOUT	Output variables
	PMMRPT	Output variables
	PMMFTAB	Reporting variables
	QONROAD	On-road distillate quantity, conversion factor
	PONROAD	On-road distillate price
	LFMMOUT	Output variables
Coal Market	COALOUT	Output variables
	COALREP	Reporting variables
	COALPRC	Electric power sector coal prices at the coal demand region level
	USO2GRP	Coal output by emission categories for Electricity Capacity Planning interface

Pyfiler

NEMS2023 introduced PyFiler, which allows GDS variables to be shared between Python and Fortran programs in memory using NumPy’s F2py library. F2PY facilitates creating/building native Python C/API extension modules that make it possible to call Fortran from Python. This interface enables the fast, seamless transfer of data between the Python integration code and the legacy Fortran module code in NEMS. We significantly expanded PyFiler in NEMS2025, so it now serves as an access point for the NEMS GDS. In order to work with PyFiler, the NEMS Fortran code is now compiled as a library for Python rather than as a standalone executable.

PyFiler is used to support most reads and writes out of NEMS.

Energy market data representation

The Energy Market Data define the energy quantity and price variables for NEMS. These variables are the principal values subject to convergence testing in the integrating algorithm. The Energy Market Data are part of the NEMS Global Data Structure and are stored in the following blocks:

• QBLK	Energy consumption quantities by fuel and sector
• MPBLK	Energy prices by fuel and sector, excluding any CO_2 fees in effect
• AMPBLK	Energy prices by fuel and sector, including any carbon dioxide fees in effect
• MXQBLK	Expectations for energy consumption quantities
• MXPBLK	Expectations for energy prices

The quantity and price structure does not attempt to represent all energy flows, but instead it focuses on the primary variables needed to design the NEMS equilibrating methodology. In addition, the Energy Market Data structure defines the fuel and sectoral energy classification for the NEMS energy balance .

In general, the energy prices match the corresponding consumption quantities . The exceptions include:

- Detailed refinery sector prices are omitted even though refinery fuel quantities are included because the projections don’t require refinery sector prices to be separate from the rest of the industrial sector. The industrial fuel prices are the delivered prices to industrial fuel consumers, including refineries. As a result, the industrial sector prices match the coverage of the corresponding industrial consumption quantities.
- Prices for some industrial petroleum categories are combined in the industrial *Other petroleum* category to eliminate unnecessary detail. That is, the industrial *Other petroleum* price is defined as the average price of three consumption categories: still gas, petroleum coke, and other petroleum. The *Other petroleum* price is not needed by any NEMS module but is required for reporting purposes to determine the average price of all petroleum products.

Delivered prices for renewable energy categories are left undefined because there are no meaningful market prices for them. For example, no delivered prices are associated with hydroelectric, geothermal, wind, solar thermal, and photovoltaic energy sources. In the case of biomass, supply curves for four different feedstocks (forestry residues, urban wood waste and mill residues, agricultural residues, and energy crops) are generated for the Liquid Fuels Market Module and the Electricity Market Module, and a composite average price is calculated.

NEMS uses variable names for consumption quantities and prices, along with a two-character product code mnemonic for each product. Each array is a two-dimensional, floating-point array. The first dimension represents the nine census divisions as well as a tenth position that is blank and an eleventh position reserved for the national total. The second dimension represents 61 years from 1990 to 2050. Quantities are stored in trillions of British thermal units (Btu). Prices are stored in 1987 dollars per million Btu, as deflated by the chain-weighted price deflator for gross domestic product.

A related part of the Energy Market Data structure is made up of the variables that hold energy market expectations. The Integrating Module maintains a separate set of arrays to store consumption and price expectations. The expectations arrays are updated according to the foresight options under consideration. The expectations arrays are defined like the standard energy market arrays, each with an additional leading character, *X*. Not all fuel price and demand quantity detail is represented in the expectation arrays.

Restart file

At the beginning of a run, the Integrating Module reads initial values for all data in the Global Data Structure from a user-specifiable version of a special file, called the Restart file. The Restart file contains a starting point for the case under consideration, consisting of results from a previous simulation. During the run, much of these data are updated and changed. For example, alternative values for key module parameters and input assumptions, read separately from the user interface file or other sources, override the values stored in the Restart file. At the end of the run, a new Restart file is created with all the data from the run. The file is available for future runs, as well as to link with reporting and database management routines.

The restart file promotes modularity by supplying values for all shared variables, regardless of whether the module that creates them is active in the run. Prices, quantities demanded or supplied, and other variables normally generated by a module that is switched off for the current run are provided instead by the Restart file.

NEMS2023 is in the midst of the transition between the legacy unformatted (.unf) data file and the npz data file that will be used in future NEMS versions.

The global data are separated into groups of variables known as *blocks*. The NEMS modules may access data from, and write results to, the block variables once the data are loaded into memory.

4. Integrating Module Solution Methodology

The Integrating Module contains the *converge* submodule, which implements the NEMS solution algorithm. The algorithm relies upon consecutive execution of the NEMS component modules iteratively to achieve energy market equilibrium for each projection year. Using the NEMS Global Data Structure as its inputs, the converge submodule tests whether convergence has occurred, and it optionally adjusts the solution values to aid the convergence process.

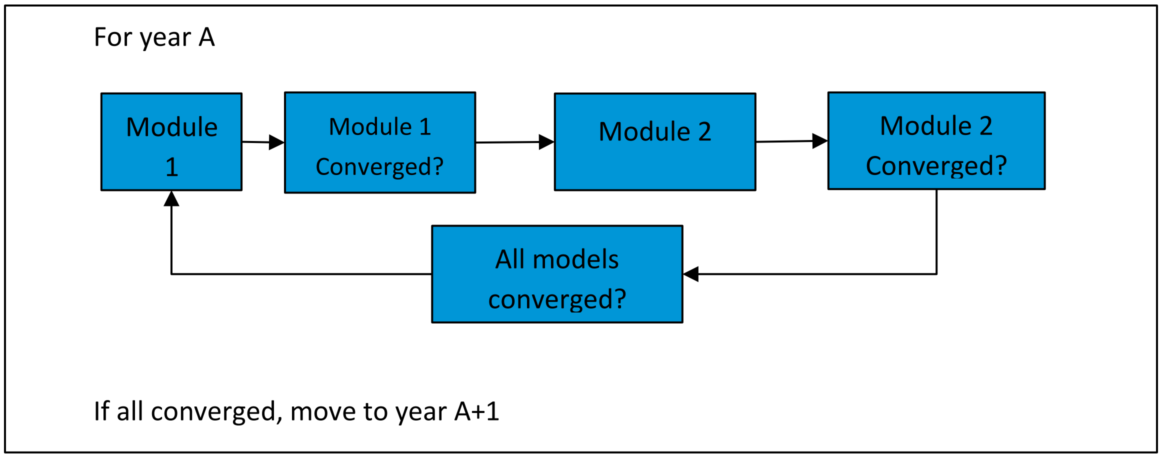
Within the converge submodule, there are two convergence tests for a cycle, and for an iteration.

The NEMS iteration

Introduction

The iteration solution is the inner loop of NEMS, and where NEMS iterates each modul4 over each year repeatedly before going to the next. Each module is checked for convergence before the next module runs, and then relaxation is applied, if appropriate.

Figure 2: Simplified representation of the iteration loop



The modules in NEMS represent the demand, supply, and conversion segments of the energy market as well as modules to provide economic, international market, and other feedbacks. In effect, these modules represent energy supply and demand curves. That is, the supply and conversion modules determine prices and sources of supply, given the quantity of fuel demanded. The demand and conversion models determine the fuel demands, given the prices of those fuels. The solution algorithm attempts to determine a vector of fuel prices and quantities so that supply and demand curves in all fuel markets equilibrate. That is, a solution occurs when energy demands and prices, along with the macroeconomic variables, reach stable, convergent values.

The Iteration Solution algorithm

To reach a solution for each iteration, the convergence submodule solves simultaneous equations implied by the supply, demand, and conversion modules. The approach applies the Gauss-Seidel algorithm, which solves a set of simultaneous equations. Gauss-Seidel is an iterative method of solving simultaneous linear equations by replacing the independent variables with their previous solved-for values. Although equations within NEMS can be non-linear, this method is expected to provide an equilibrium solution because the equations are either monotonically increasing (as are the supply curves) or monotonically decreasing (as are the demand curves).

In effect, the approach groups the equations and variables into subsets. For NEMS, the subsets consist of predefined fuel supply, energy conversion, and sectoral demand modules. Each subset of equations is solved, keeping the other variables constant at their trial values and ignoring the effects of current variables on equations in other subsets. The process is repeated for each subset, updating the trial values for each variable from the previous solution.

More formally, for a stylized NEMS, the nonlinear system of equations could be represented by

$$x_i = g_i(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n) \quad \text{for} \quad i = 1, \ldots, n, \tag{1}$$

having the market clearing or equilibrium solution vector

$$x = (x_1, \ldots, x_n).$$

The solution process assumes a set of initial values, denoted x^0, where

$$x^0 = (x_1^0, \ldots, x_n^0).$$

A trial solution for iteration *k* for a certain year is denoted by x^k, where

$$x^k = (x_1^k, \ldots, x_n^k).$$

Each $g_i(x)$ uses one or more of the elements of the trial solution vector x^k, excluding its own solution, x_i^k.

A solution iteration *k* begins with the evaluation of g_1 and continues solving each g_i, ending with g_n. The solution of g_i in iteration *k* updates the solution estimate to

$$x = (x_1^k, x_2^k, \ldots, x_{i-1}^k, x_i^k, x_{i+1}^{k-1}, \ldots, x_n^{k-1}) \quad .$$

The updating process continues until an iteration-*k* trial solution is derived for all x_i.

After evaluating g_i^k, the values of the solution variables are compared with the values from iteration *k-1*. A final solution, x^k, has been achieved if, after all modules have been executed, the absolute values of the proportional changes in the x_i remain smaller than a specified tolerance, ε :

$$\left| \frac{x_i^k - x_i^{k-1}}{(x_i^k + x_i^{k-1})/2} \right| < \varepsilon$$

for *i* = 1, ..., *n*. Values of ε can be chosen on a variable-specific basis. The typical values used are in the range of 1% for the census division variables, less for the national macroeconomic variables. In the convergence tests, the denominators use an average to avoid convergence difficulties if either the starting value or a trial solution value is equal to zero.

After the convergence criteria have been met, another iteration is performed to test whether the solution is stable and to allow the modules to perform final processing for the projection year. As a result, the final converged solution vector for the projection year is x^{k+1}, where k is the first iteration for which the solution meets the convergence criterion.

A procedure referred to as *relaxation* is used to control the equilibration process and aid in resolving some convergence problems. If the relaxation option is selected, changes in values of convergence variables between iterations are dampened by a user-specified factor. The selection of appropriate relaxation parameters may speed convergence and lead to a more stable and robust solution process. The relaxation *assignment statement* is of the form:

$$x_i^k = x_i^{k-1} + r_i^k\,(x_i^k - x_i^{k-1})$$

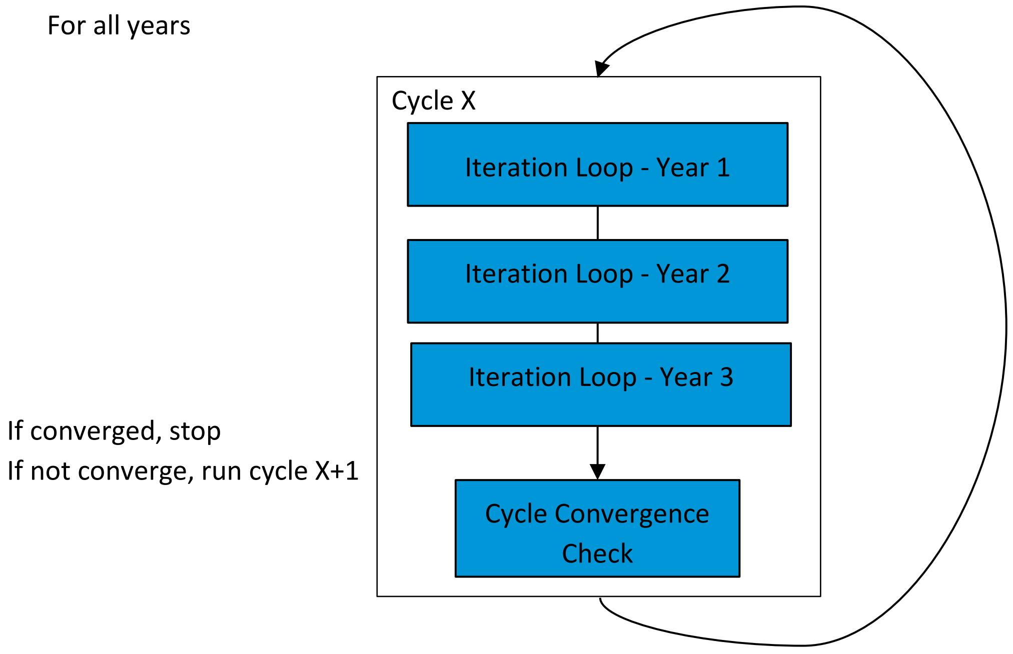
where r^k_i = relaxation factor for a convergence variable *i* for iteration *k*. Note that the specification of relaxation factors is variable specific and iteration specific. The module can specify varying relaxation fractions, depending on the iteration number, as an option. This feature is used to allow greater dampening after the first few iterations. Convergence parameters, including the tolerances and relaxation fractions for each variable, are specified through the input file *mncnvrg.txt*.

To handle cases where the procedure does not converge on a solution or does not achieve the specified tolerance, a limit on the number of iterations terminates the algorithm for the current projection year. In such cases, the model performs the additional iteration mentioned in the previous paragraph, reports the convergence status with a list of the variables failing to converge, and then proceeds to the next projection year. The final solution for the projection year is, therefore, the result one iteration beyond the non-converged trial solution.

The NEMS cycle

Introduction

The *cycle* solution is the outer loop of NEMS, and allow NEMS to solve with perfect foresight structures. Each cycle involves the iterative execution of all of the projection years.



Solution values for successive cycles are compared to determine if expected values (from the previous cycle) and realized values (from the current cycle) converge. A program performs the intercycle convergence checks and scores the degree of intercycle convergence using a qualitative metric (discussed more below). It is typical, then, to run NEMS in sets of 4 or more cycles to achieve intercycle convergence. In addition, a relaxation procedure, similar to the single-year relaxation procedure, can be applied to speed up convergence between cycles. Parameters for testing convergence between cycles are separate from those for testing convergence between iterations.

The cycle solution algorithm

A qualitative metric for convergence is presented in a NEMS output report (NEMS report writer output Table 150) as an aid in evaluating the degree of convergence. The convergence metric, known as the Grade Point Average (GPA), scores the convergence tests on a four-point, academic-style grading scale. With this idea, a run's convergence status is revealed with a single number associated with a sense of quality: a 4.0 GPA is a straight A average, for example. A run with a convergence GPA of 2.0 (a *C*) is average, while a GPA of 1.0 (a *D*) is a poor grade. This heuristic grading scale is derived using a weighted average of the absolute value of percentage differences in convergence variables, aggregated across sectors and regions. The convergence GPA is calculated as follows:

1) Compute deviations for convergence variables for each fuel, region, and sector in year. Let:

- $DEV_{f,r,s,y}$ = Absolute value of deviation in a convergence variable: fuel *f*, region *r*, sector *s*, year *y*, where a deviation is one of the following:

    a. Quantity deviation: absolute value of (the current quantity minus the previous quantity)

    b. Price deviation: absolute value of the current expenditure (that is, price times quantity) minus the previous expenditure (the expenditures exclude any permit price adders)

    c. Emission allowance price deviation: absolute value of the current allowance price minus the previous allowance price.

    $PREV_{f,r,s,y}$ = Previous value for a convergence variable: fuel *f*, region *r*, sector *s*, year *y*

2) Group the convergence variables into five categories, *c*:
    a. End-use sector energy consumption quantities
    b. Electric power sector energy consumption quantities
    c. End-use sector energy prices
    d. Electric power sector energy prices
    e. Environmental permits/allowance prices: carbon dioxide, sulfur dioxide, and mercury

3) Aggregate the deviations (*DEV*) across regions, fuels, and sectors within each of the five categories, *c*, and express the deviations as percentage of the corresponding previous values (*PREV*). Let $AC_{c,y}$ = the aggregated change (or deviation) for category *c* and year *y*, expressed as a percentage. That is,

$$AC_{c,y} = \frac{\sum_f \sum_r \sum_s DEV_{c,f,r,s,y}}{\sum_f \sum_r \sum_s PREV_{c,f,r,s,y}} * 100,$$

where the sums are over all fuels *f*, regions *r*, and sectors *s* that belong in category *c*.

4) Compute a composite score by averaging the aggregated changes (AC) of the five categories, using the following weights (the basis for the values is described further below).

Table 2: Convergence variable weights by category

Category	Weight
End-use sector energy consumption quantities	24.5
Electric power sector energy consumption quantities	24.5
End-use sector energy prices	24.5
Electric power sector energy price	24.5
Environmental allowance fees	
Carbon dioxide (if applicable)	0
Sulfur dioxide	1
Mercury (if applicable)	1

5) Scale or grade the composite score into a grade point average (GPA) by interpolating the score from the following table:

Table 3: Composite score to GPA

Score (percentage basis)	Grade on four-point scale	Letter grade
0.5 or less	4.0	A
2.0	3.0	B
5.0	2.0	C
10.0	1.0	D
15.0 or more	0.0001	F

This process is also used to calculate the metric, based on national-level data.

The weights and the grading scale tend to magnify the importance of common convergence problems. The carbon dioxide allowance price has been weighted as zero (so, not entering into the convergence decision) because the sectoral prices include the carbon dioxide allowance price; so, any movement from cycle to cycle will be reflected in the end-use prices. This allowance price also has a significant effect on capacity expansion decisions made in the electric power sector and macroeconomic feedbacks, so stability in this price is essential for inter-cycle convergence. Fuel demands and prices in the electric power sector are also given a relatively strong weight in the scoring. Flexibility in electric power sector fuel demands, the use of linear programs for plant dispatch and capacity build decisions, and complex interactions with the coal supply module with respect to environmental constraints all tend to foster convergence difficulties in this sector. The capacity build decisions are influenced by fuel price expectations and any energy-related taxes or emission allowance fees. These capacity choices, along with the decisions in the fuel dispatch submodule, help determine electric power sector fuel consumption and can become a primary source of inter-cycle convergence problems.

The NEMS cycle runs continue for a user-specified number of cycles or until the inter-cycle convergence objective has been met. The objective is based on the average of the three lowest yearly GPAs. If this metric is lower than the user-specified minimum, the cycling continues. Otherwise, the cycling stops. Additional user-specified options can be set to perform all of the requested cycles regardless of convergence or to perform at least a certain number of cycles.

Parallel NEMS

Instead of running all the NEMS models sequentially, NEMS can be run in two parallel partitions. Modules are grouped together, reducing the number of parallel processes, by using a combination of the Jacobi and Gauss-Seidel methods. The relative lack of connectivity between the electric power sector and the refining industry allows for the following grouping of related modules:
Partition 1:

- Liquid Fuels Market Module
- International Energy Module
- Hydrocarbon Supply Module
- Natural Gas Market Module
- Macroeconomic Activity Module
- Residential Demand Module
- Commercial Demand Module
- Transportation Demand Module
- Industrial Demand Module
- Carbon Capture, Transportation and Sequestration Module

Partition 2:

- Electricity Market Module
- Coal Market Module
- Renewable Energy Module
- Residential Demand Module
- Commercial Demand Module
- Hydrogen Market Module

After these two processes complete, the results are merged together, and another cycle is run.

Foresight approach

Several modules simulate planning decisions to acquire additional capacity that will be required in future years. These include the Electricity Capacity Expansion submodule, the pipeline capacity decisions for natural gas in the Natural Gas Market Module, and the refinery capacity decisions in the Liquid Fuels Market Module.

To simulate such decisions, information on future demands and prices must be assumed. Although each module solves one projection year at a time, their simulations of planning activities involve an extrapolation of energy market conditions. Those modules simulating new capacity construction decisions apply an assumption about foresight in their expectations of future energy prices and quantities. In NEMS, a set of price and quantity variables is defined to store expectations. For $\hat{y} > y$,

$XP_{f,s,r,\hat{y}}$	=	Expected prices of energy products beyond the current projection year
$XQ_{f,s,r,\hat{y}}$	=	Expected consumption of energy products beyond the current projection year

The foresight mode determines how the expectation variables are calculated. Under myopic foresight, the expected values are simply held constant at their current trial values. For adaptive expectations, the Integrating Module calculates minor extrapolations of present-year conditions. Foresight is, therefore, always calculated by looking forward to the consequences of conditions in the present iteration year, not by attempting to reach some end state determined *a priori*. The treatment of expectations is discussed in greater detail under *Expected Value Foresight*.

In terms of the energy market interactions, the sectoral demand models estimate current-year energy demands $Q_{f,s,r,y}$ and energy-related capital stock additions as functions of current and expected energy prices. The supply modules estimate end-use prices $P_{f,s,r,y}$ and capacity additions as functions of current and expected energy demands. The conversion modules (electricity and refinery) are viewed primarily as supply components, but they represent both consumers of primary energy and suppliers of energy products.

For some model components, a rational expectations, or *perfect foresight* approach, is used implicitly or explicitly. Where these approaches are used, expectations for future years are defined by the realized solution values for these years in a previous run. This approach is used, for example, for the energy demand expectations used for capacity planning of energy infrastructure (pipelines and refineries). The other area is for market-based approaches to limit carbon dioxide emissions, where knowledge of future emission taxes or permit prices is assumed to be known in advance.

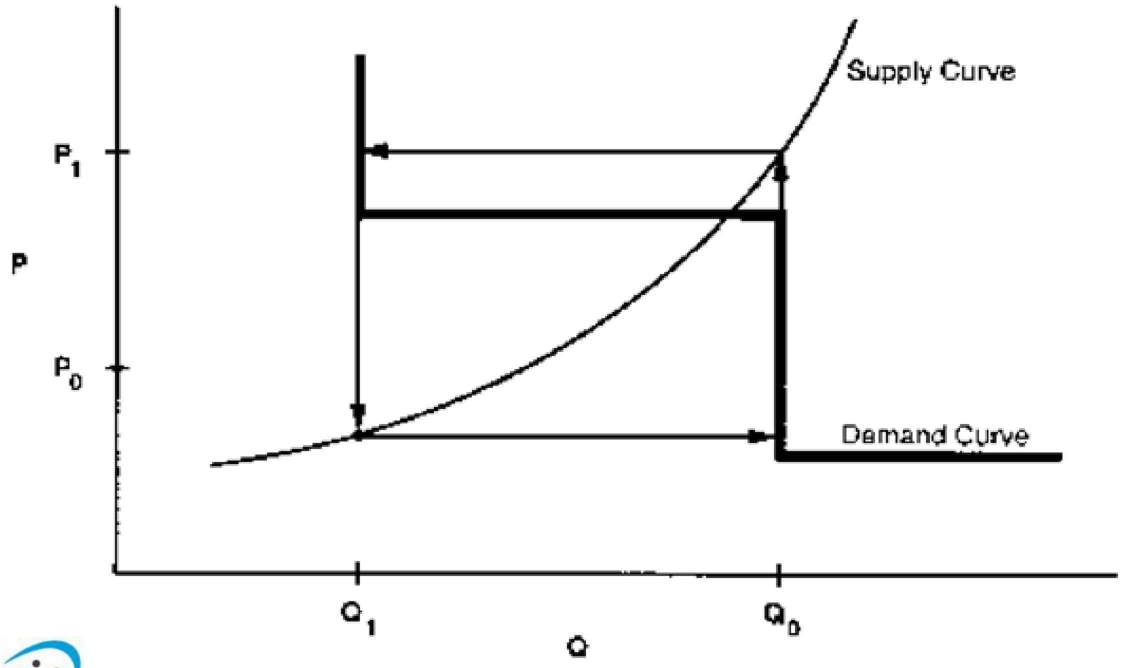
Discontinuities and convergence problems in NEMS

The characterization of NEMS as a set of supply and demand curves provides a useful framework for discussing convergence properties. Although supply and demand curves are generally treated as continuous functions, various NEMS modules contain linear programs or their analogues that result in discontinuities. Such discontinuities cause significant problems in the solution process.

Several modules incorporate algorithms that yield these discontinuous results. For example, the International Energy module outputs a set of crude oil supply curves and petroleum product import supply curves that the Liquid Fuels Market Module translates to step curves for input to a linear program, representing refinery operations and solving for fuel prices and refinery fuel demands to minimize costs. This type of approach yields discontinuous petroleum pricing and fuel demands. The Electricity Fuel Dispatch submodule is also implemented as a linear program and contains discontinuities as a result of the nature of the merit-order plant dispatch. The coal distribution submodule is also a linear program. So, each of these modules introduces discontinuities into the NEMS solution process.

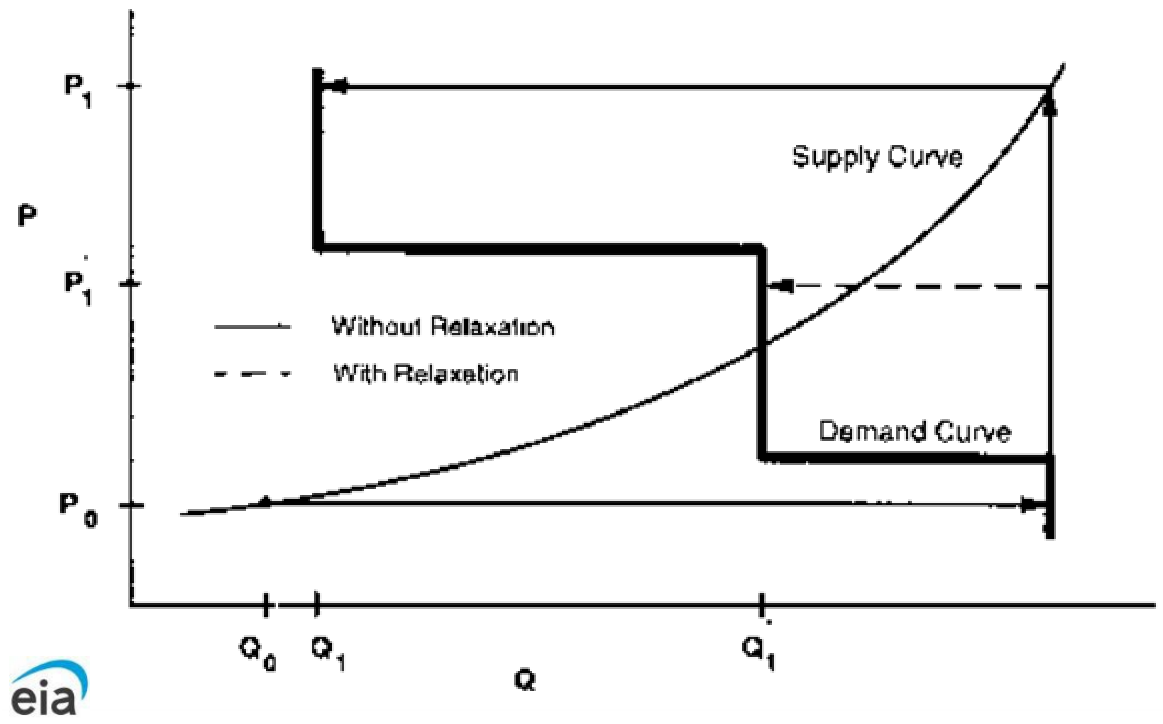
You can see the effect that having discontinuities has on the solution process by using step-function demand curves with continuous supply curves. The same conclusions may be drawn as long as either or both of the supply and demand curves are step functions (Figure 3 and Figure 4).

Figure 3. The supply curve cuts across the horizontal portion of the demand curve



Data source: U.S. Energy Information Administration

Figure 4. The supply curve cuts across the vertical portion of the demand curve



Data source: U.S. Energy Information Administration

The supply curve determines the price used in the demand curves, and the demand curve then provides a quantity (Figure 3 and Figure 4). The solution path resulting from applying the Gauss-Seidel algorithm is delineated by arrows: a horizontal arrow shows the quantity response from the demand curve, and a vertical arrow shows the price response from the supply curve.

When the supply curve intersects the horizontal portion of the demand curve, an oscillation in the solution between quantities Q0 and Q1 and prices P0 and P1 occurs (Figure 3). When the intersection of the supply and demand curves is on the vertical portion of the demand curve, you can achieve equilibrium with the Gauss-Seidel algorithm using relaxation, even if the unrelaxed algorithm yields an oscillation in the solution (Figure 4). Figure 3 has no relaxation fraction, *r*, for which convergence will occur. However, a value for *r* can be found so that the oscillation occurs in no more than two steps. Provided the steps are small enough to fall within the convergence tolerance, relaxation can prevent oscillations between steps from being a convergence problem.

Expected value foresight

Energy projections involve assessing changes in energy-using capital stocks and choices among energy supply alternatives. This analysis requires simulation of such decisions as the selection of durable appliances, the planning of electricity generating capacity additions, and the planning of infrastructure expansion, such as natural gas pipeline additions or E85 fueling stations. The economic evaluation of these decisions requires energy demand and price expectations for lifecycle cost and capacity addition calculations. An objective in this aspect of the modeling is to simulate such decision-making in the aggregate for predictive and analytical purposes by representing how players in the energy marketplace make long-term planning decisions, rather than by deriving the theoretically optimal long-term expansion path. As a result, formulating foresight assumptions is open to alternative approaches based on observed industry practices.

NEMS could, in principle, approach the issue of foresight by prescribing a desirable end state for the energy marketplace and calculate backwards in time to prescribe how best to arrive there. However, as a simulation, NEMS calculates foresight as an extrapolation of the present state of energy markets, subject to announced policies. Rather than determining how to arrive at the planned future, NEMS can evaluate whether present plans could result in the desired end state.

In reality, different methodologies for treating foresight are used in different sectors and supply areas, and alternative approaches to representing expectations may yield significantly different planning decisions. As a result, treatment of foresight becomes an important modeling decision.

There is no one best approach to treating foresight. The National Research Council recommended developing several options for modeling foresight.[3] As a result, an objective in building NEMS was to include the flexibility to support different approaches to foresight to allow experimentation and future modeling changes. In addition, the option to treat foresight consistently throughout the modeling system is desirable.

[3] National Research Council, The National Energy Modeling System, Washington DC: National Academy Press, 1992.

The purpose of dealing with foresight and expectations in the Integrating Module is to be able to represent different types of foresight consistently. At the same time, the Integrating Module allows individual modules to handle foresight independently if industry practice requires different approaches. To achieve this flexibility, we built each NEMS module to examine results of a centralized on-off switch to determine whether the module should use centrally generated expectations. When this central-control switch is turned on, the module uses these expectations; otherwise, the module uses self-generated expectations.

The following three methods generate expectations:

- With the *myopic expectations* option, expected prices for any projection period are assumed to be constant in real-dollar terms relative to the current period in which decisions are being made. This case generally applies to expected prices and not expected quantities because an assumption of constant energy quantity demanded is rarely assumed.
- The *adaptive expectations* (or *extrapolative expectations*) approach assumes planners extrapolate recent trends when making long-term decisions. For the system-generated expectations, this assumption about foresight is implemented by extrapolating the current projection year prices and quantities using the average annual growth during the previous few projection years. For example, the expectations generated representing 2021 for use in model year 2020 would be determined from the growth during the past few model years (for example, 2018 to 2020), and the number of years are a model option. For expectations generated within individual modules, we can use more elaborate behavioral models, or adaptive expectations.
- The *perfect foresight* approach is based on the theory of rational expectations. This approach generates an internally consistent scenario where forming expectations is consistent with the projections realized in the model. In practice, perfect foresight describes the configuration and solution algorithm that achieves the convergence of expected values and realized solution values. A variation in the integrating algorithm was required to implement perfect foresight. This option involves iterative cycling of NEMS runs, in which each cycle is a complete pass during the entire projection period. The objective is to have expected values and realized values converge between cycles, a state referred to as inter-cycle convergence, in addition to having convergence within the cycle for individual projection years, or intra-cycle convergence. As a result, it has become necessary to evaluate NEMS runs with respect to both inter-cycle convergence and intra-cycle convergence.

The Electricity Market Module depends heavily on expectations techniques and requires fuel price expectations for natural gas, oil, and coal for its capacity planning submodule. The capacity planning submodule also requires expectations for electricity demand. At present, some aspects of the oil and natural gas price expectations for the Electricity Market Module are still implemented in the Integrating Module:

- Oil product price expectations are calculated from an external projection of world oil prices, assuming a constant markup between the regional product price and the world oil price. In each projection year, the assumed markup is derived from the previous projection year:

  $P_{c+y} = (P_c - W_c) + W_{c+y}$ for y=1,...,30 years (planning horizon for power plants) where P_c and W_c are the product price and the exogenous world oil price from the previous projection year, and P_{c+y} and W_{c+y} are the prices in the expectation years.

- The wellhead price expectations through 2050 are generated by a perfect foresight method (by default). The wellhead price expectations are taken as a weighted average of the previous cycle's realized prices and its expected prices. The weight is specified by the user. Delivered natural gas prices are derived from expected wellhead prices assuming a constant markup between the delivered prices and the wellhead price.

The wellhead price expectations for the post-2050 period are based on a nonlinear function that relates the expected wellhead gas price to cumulative domestic natural gas production. Increases in cumulative production would be associated with the depletion of domestic resources and, in turn, general expectations of increases in price in the long run. The following equation tries to capture this general idea:

$$P_y = A_y * Q_y^{e} + B_y,$$

where P is the wellhead price, *Q* is the cumulative production from 1991 to future year *y* in the planning horizon, *e* is a user-specified parameter, and A_y and B_y are determined for each projection year, as explained below.

The approach was developed to have the following properties:

- Prices should be upward sloping as a function of cumulative natural gas production because prices could be expected to rise as existing resources are depleted.
- The rate of change in wellhead prices is a function of the economical resources that remain to be discovered and produced. The value of the parameter *e* determines the shape of function.

The approach assumes that, at some point in the future, a given target price, *PF*, results when cumulative natural gas production reaches a given level, *QF*. So, the target value *PF* is an assumed input to the approach, while *QF* is assigned as the resource base in the Hydrocarbon Supply Module for a specified year (2018 in AEO2022). In the *Annual Energy Outlook 2022,* the assumed value of *PF* was $9.00 per thousand cubic feet (in real 1998 dollars), corresponding to a cumulative production (*QF*) of 2,418 trillion cubic feet. The annual production is assumed to grow at the rate observed during the previous three years within the projection. The parameters of the price equation, A_y and B_y, are determined for each projection year such that the price equation will intersect the future target point. That is,

let D_{y-1}	=	previous year's natural gas production
let PS_{y-1}	=	previous year's wellhead gas price
let QS_{y-1}	=	previous year's cumulative natural gas production since 1991

$$A_y = (PF - PS_{y-1}) / (QF^{e} - QS_{y-1}^{e})$$

$$B_y = PF - A_y * QF^e$$

The following assignment statement extrapolates cumulative production for future years, *y* = 1, ... , 30 years (with 30 years being the maximum planning horizon for power plants):

$$Q_y = Q_{y-1} + D_{y-1}$$

This generates the expected wellhead prices:

$$\begin{aligned} P_y &= A_y * Q_y^e + B_y \\ &= PF + \left(Q_y^e - QF^e\right) * \left(\frac{PF - PS_{y-1}}{QF^e - QS_{y-1}^e}\right). \end{aligned}$$

5. Managing NEMS runs

System Design

RabbitMQ is an open-source message passing broker running the Advanced Message Queuing Protocol (AMQP). Celery is a Python-based system for setting up and running task queues that can use RabbitMQ as the message broker.

The NEMS job queue has the following features:

- A Celery client, called by the user, which handles the initial model setup for the NEMS run. After setup, the Celery client sends a message through the broker to the worker via the Celery task command.
- Celery workers on each of the three dedicated servers, to perform the actual execution of the NEMS runs. Celery workers act as consumers of messages from the RabbitMQ Server.
- A run monitor accessible by users. This monitor shows the status of each run, including user, scenario name, datekey, part (if applicable), host, cycle number, year, iteration number, status, and output directory. The monitor maintains data for a configurable time – by default, it outputs files with two-day and one-week retention periods – and is searchable and sortable.

We have stood up a RabbitMQ broker with queues for processing NEMS run job requests. The queue structure is discussed in the following section. We have also developed Python worker and client scripts to wrap around the current NEMS scripts and initiate a NEMS run, along with a run monitor script to generate files which users can read to track run progress.

The key elements of the resulting system are thus:

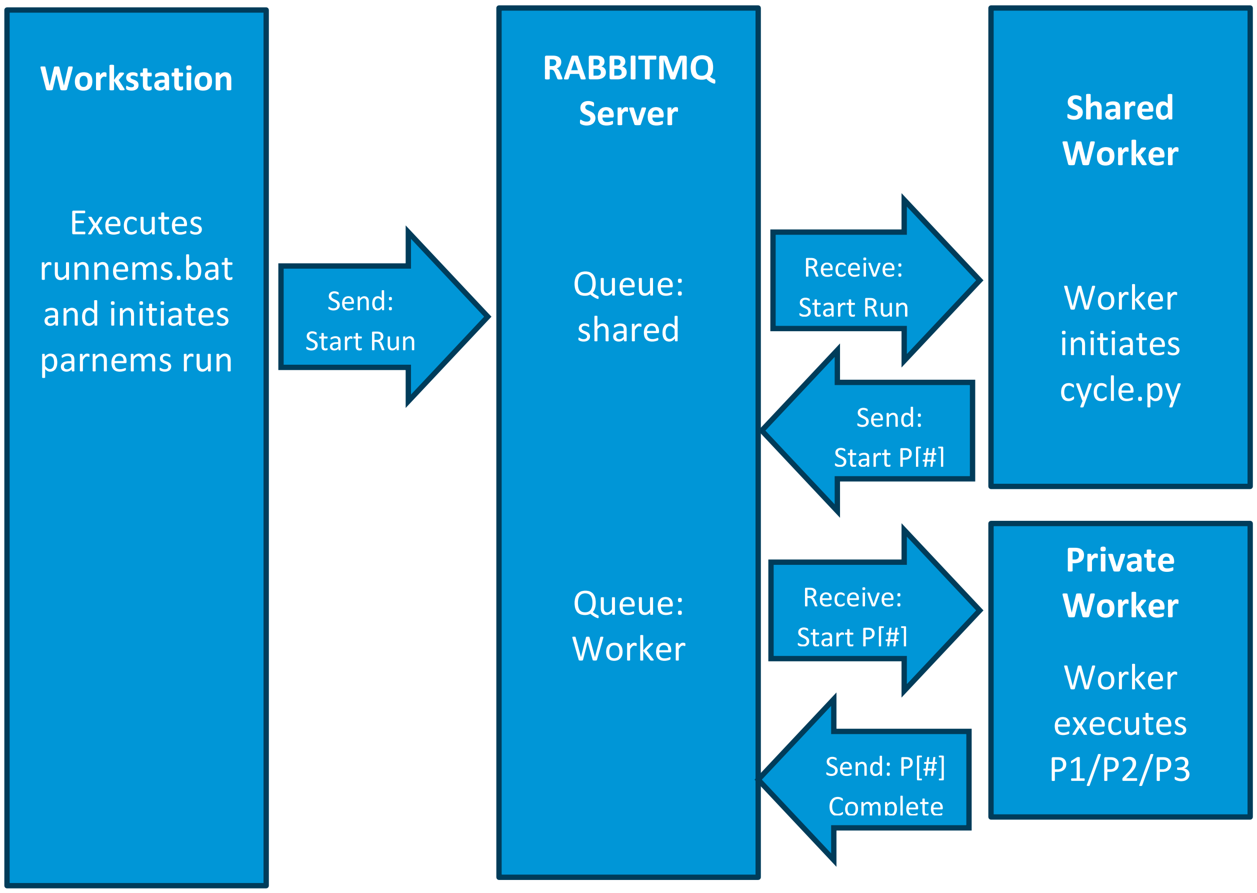
- **Run Monitor Front End:** HTML files which can be opened in any web browser to display a searchable, sortable list of ongoing and completed runs.
- **Run Monitor Back End:** A continuously-running Python script which regularly regenerates the aforementioned HTML files to remain up-to-date with run status.
- **RabbitMQ Server:** A broker service running the AMQP protocol. The Celery clients submit messages to this server to request execution of a NEMS run. Messages are made available to the connected Celery workers on the dedicated NEMS machines to execute the run, distributed by workload and availability.
- **Celery Workers:** Celery workers are to be deployed on the three dedicated servers, where actual execution of NEMS runs will occur. Celery workers act as consumers of messages from the RabbitMQ server.

Queue Structure

NEMs may be executed in either of two ways, either as a single executable or loosely coupled parallel structure. In the former incarnation (jognems), each module is called in sequence until the run is completed. In the latter, only a subset of modules is executed in each of two partitions (P1 and P2). The output of P1 and P2 is then merged into a third partition (P3).

When designing the structure of queues through which tasks would flow, it was desired that each part of a parallel NEMS (parnems) run should be executed as its own task. Accordingly, and with the aim of maintaining similarity in execution between parnems and jognems runs, we devised the following arrangement (using a parnems run as an example):

Figure 5. Queue Process Flow



Each machine that executes NEMS runs operates two Celery workers to receive tasks. One worker listens to a queue named “shared” and used by all workers, and worker listens to the queue named for the COMPUTERNAME environment variable of its host machine. All runs will initially be sent to the “shared” queue, from which they can be fetched by any worker based on availability.

Once a worker has received the overall task for a run, it is necessary for all tasks within that run to be managed by that same worker, so that there is no need to repeatedly transfer run files among machines. As such, in the case of a parnems run, the worker will send tasks for parts 1 and 2 of the run (P1 and P2) to the COMPUTERNAME queue, on which only that worker is listening. This ensures that the same worker will receive and execute the P1 and P2 tasks. Once these are completed, a part 3 (P3) task will be sent in the same way. In the case of a jognems run, instead only a single task will be sent to the COMPUTERNAME queue for the entire cycle. In either case, the process will then be repeated as many times as required by the parameters of the run.

In addition to its utility for run-monitoring and similar purposes, this design can easily be adjusted to allow subtasks to be shared across some or all of the relevant machines, in case a future system will have runs executed on shared drives such that file transfer overhead will no longer be a concern. If a worker's setup is adjusted such that it is also listening to the COMPUTERNAME queue(s) of one or more other machines, it will be permitted to execute any subtask sent by those computers. Alternatively, with appropriate code changes, it would also be possible to bring all workers exclusively onto the shared queue, allowing any one machine to receive tasks sent by any other. These possible changes will be discussed in more detail in the following section.

Cycle.py

The script cycle.py executes with a celery message to run nems_flow on a workers . The Python script run_task.py is used to send the message to the COMPUTERNAME queue.

Tasks.py

The Python program tasks.py first establishes a connection with the RabbitMQ server. It then defines the Celery task exec_at_loc, which is used by the worker to execute cycle.sh and nems.exe. This task is a function which takes three parameters: userid (User ID), loc (location), and comm (command). The user ID is only passed to facilitate tracking by the run monitor and is not actually used by the task itself.

First, exec_at_loc checks if the location provided is in D:/workdir. If it is, then the script simply executes the command provided at that location. The task then waits for this subprocess to complete and returns a code reporting its completion status.

If the location provided is not in D:/workdir, this indicates the selected output directory for the NEMS run is the one that was created in the initial setup by nems_setup.py. Accordingly, scenario and datekey are determined using the final elements of this path and all files are copied from it into D:/workdir/[scenario]/[datekey]. This folder is created if it did not exist or replaced if it did (though the latter should never be necessary, as all scenario-datekey pairs should be unique). The command provided is then executed at this new location, as above. Afterwards, cleanup is performed: the scedes.all file, all input folders, all .dll files, and all .exe files are removed from the new location if present, though if ftab.exe exists it is first copied to ftab.xxx. The contents of the cleaned-up folder are then copied back to the originally-provided location, overwriting where necessary, and then deleted from D:/workdir once the copying is complete. Finally, as before, the task returns the return code of the subprocess responsible for command execution.

Worker_start.bat

The worker_start.bat batch script launches a Celery worker. It first ensures that the correct Python environment is active, so that the necessary packages to run the worker will be available, and then launches a worker using tasks.py. It is also set to use a pool of threads to execute a maximum number of concurrent tasks; once that limit is reached, no new tasks will be accepted until a slot is freed up by a task completing.

run_task.py

The Python program run_task.py provides a single function, run_task. This function takes three arguments: loc (location), comm (command), and q (queue). It establishes a Celery connection to a

RabbitMQ server – currently configured to be on ASHTSTNEMVIR002 with username "user" and password "test". It then sends a message to the specified queue to run exec_at_loc, from tasks.py, using the provided location and command. It also gets the value of the USERNAME environment variable and passes this to exec_at_loc; again, this is for tracking purposes only, as exec_at_loc does not actually use user ID for any direct purpose. The results of this task are then returned.

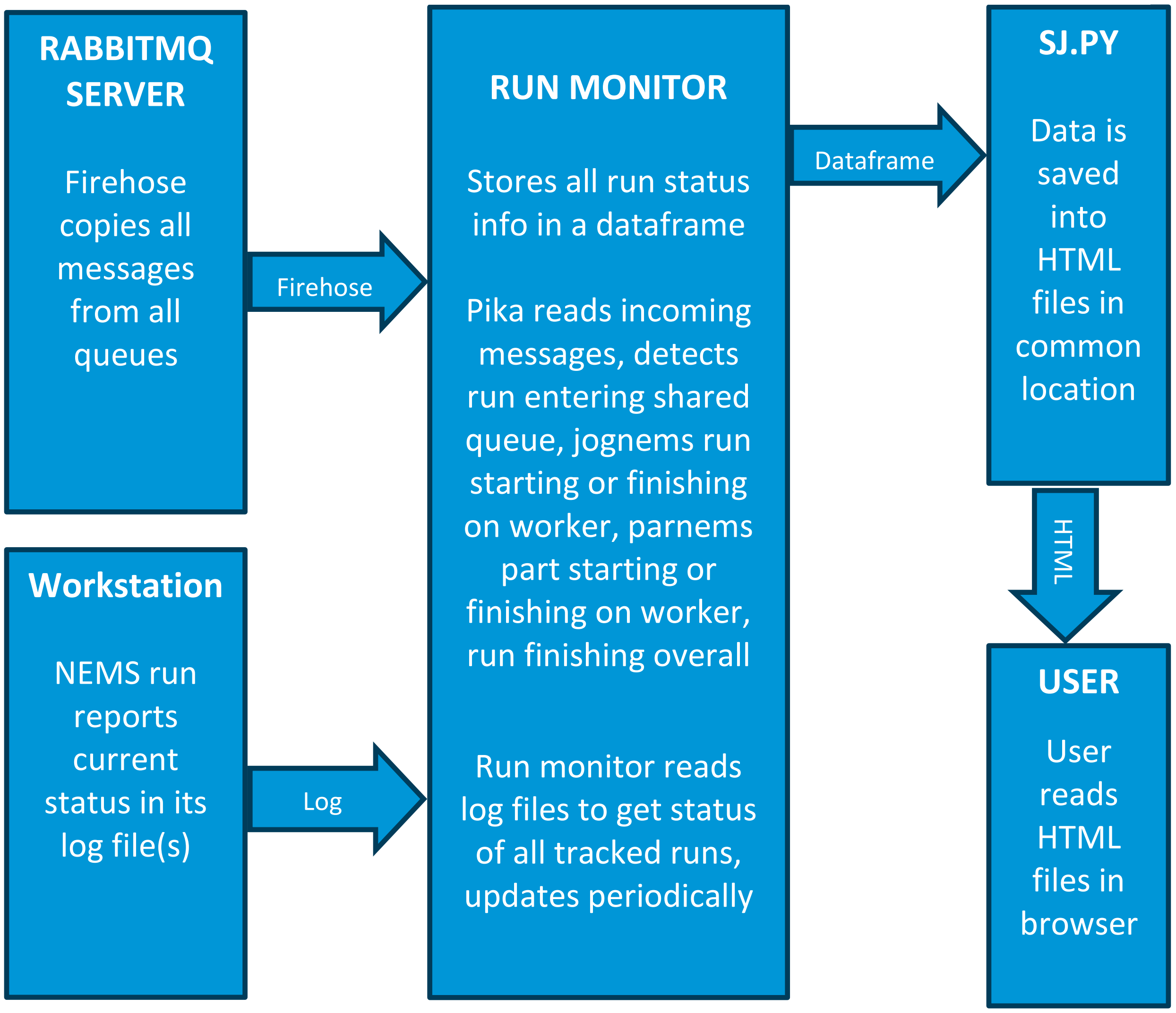
If run_task.py is launched as a Python script directly, then it will execute run_task using the command line arguments provided to it as input. In this case, it will complete with a return code equal to the results of this function. Since the run_task returns the results of the task, and the exec_at_loc returns the return code of the subprocess used to execute its provided command, this will propagate any errors in command execution back to the place where run_task.py was called.

Run Monitor

The run monitor is composed of a pair of HTML files – one for records up to two days old, one for those up to one week old – which are generated and regularly updated by a Python script kept continuously running to monitor for changes. This script takes its input from two primary sources. For tracking when runs begin or end, it gets copies of all messages in all queues from the RabbitMQ server, using its "firehose" feature. For tracking the status of these runs while they are in progress, it reads the nems_run_status_log.txt file generated by nems.exe, each line of which specifies the module, cycle, year, and iteration, with the final one being the most current.

This is achieved through the use of three scripts: eventmonitor_start.bat, nemseventmonitor.py, and sj.py. The program eventmonitor_start.bat launches nemseventmonitor.py, which processes the aforementioned input into a dataframe. The dataframe is passed to sj.py for conversion into HTML output. Taken all together, the resulting process flow is shown in the figure below:

Figure 6. Show Jobs Process Flow


RABBITMQ SERVER
Firehose copies all messages from all queues
Firehose
Workstation
NEMS run reports current status in its log file(s)
Log
RUN MONITOR
Stores all run status info in a dataframe
Pika reads incoming messages, detects run entering shared queue, jognems run starting or finishing on worker, parnems part starting or finishing on worker, run finishing overall
Run monitor reads log files to get status of all tracked runs, updates periodically
Dataframe
SJ.PY
Data is saved into HTML files in common location
HTML
USER
User reads HTML files in browser

Much like worker_start.bat, eventmonitor_start.bat is a simple script which activates the correct Python environment and then runs nemseventmonitor.py. It provides a single argument to nemseventmonitor.py: the desired location for the output HTML files.

The Python program nemseventmonitor.py was adapted from the World Energy Projection System (WEPS) event monitor. It consists of two threads: one watches for incoming messages from the firehose, while the other periodically checks for updates to the status log file of every currently-tracked run in progress. Both threads update a shared dataframe and send it to sj.py for conversion to HTML output. This output is written to a location specified by the first command line argument given when launching the script.

To track incoming messages, a Pika connection is established to the RabbitMQ server. A queue "trace" is established to receive messages from the exchange "amq.rabbitmq.trace", which is where the firehose publishes its message copies. A processing function is then set to execute whenever a message is received in the queue. This function logs the received message to a file events_log.txt in the output directory, then updates the dataframe based on its contents. Messages are sorted by their header info into five groups: no update needed, task added to queue, task started, task succeeded, and task failed.

For tasks added to the queue, the message body is parsed to determine user ID, scenario, datekey, part (if applicable), and run folder. Additionally, a check is performed to determine if this task is a subtask of an existing run. If it is, then the user ID is updated to that of the parent task, as subtasks will initially have the user ID of the user who launched the Celery worker instead of the user who launched the run. If the task is a subtask, a flag will also be set to hide its parent task in the run monitor to avoid displaying redundant rows. In either case, the dataframe is updated with a row containing the extracted values, a host name of "Pending", a status of "In queue", and a timestamp of the current time. If a row already exists with the same run folder, that row is overwritten; otherwise, a new row is added. The dataframe is then sent to be exported in HTML form.

For tasks started, the process is very similar. The host machine name is also obtained from the message body, as is the message ID. The latter is not displayed and is used only for internal tracking of which messages are associated with the same task. The Status attribute is written as "Running" rather than "In queue". Otherwise, all steps are as described for tasks added to queue.

For successful and failed tasks, only the host name and message ID are obtained from the message body. A check is performed to determine if this task is a parent to any subtasks – if so, the flag to hide it in the run monitor is unset, and instead all of its subtasks are flagged to be hidden. Then the message ID is used to find a matching row, which is updated with the extracted values and a timestamp of the current time, along with the status of "Finished" (for successful tasks) or "Failed" (for failed tasks). A row should always be found, since a task cannot succeed or fail without first being started; however, if none is, a new one will be generated with "None" in all remaining columns. Finally, a manual call is made to the function which checks for status log updates – only currently-running tasks are checked for updates, so this ensures that all values are correct before they become locked in. The dataframe is then sent to be exported in HTML form.

The thread responsible for tracking status log files runs a function to check for updates once per minute; this time can be increased or decreased by adjusting the value passed to time.sleep. For each row in the dataframe with status “Running”, the function uses the host name and run folder recorded for that row to determine where the task in question is running, then looks for a status log file named “nems_run_status_log.txt” in that directory. If this file is found, its last line is extracted and parsed for cycle, year, and iteration, which are then used to update the relevant row of the dataframe. Once all rows have been checked, the dataframe is sent to be exported in HTML form.

When the dataframe is sent for export, it is first cleared of all messages older than the past week. A more recent copy is then made containing only data from the past two days, though this does not affect the overall data stored. The former behavior can be adjusted by editing the function remove_old_messages, while the latter can be adjusted by passing it a different argument when creating the copy. Copies are then made of both dataframes without information which does not need to be displayed – rows flagged for hiding, the flag which determines this, message ID, timestamp, and parent task. These copies are sent to sj.py for conversion into HTML files in the output folder.

sj.py contains a function, generate_html_from_dataframe, which takes a dataframe and generates an HTML file based on its contents. The file displays the same columns as are contained in the dataframe. Clicking on the header of any of these columns sorts the display by that column, and there is a search box for the first column, which will be user ID unless any changes are made. Reloading the page will clear all sorting and filtering.

Operation

First, an appropriately-configured RabbitMQ server is required. In order to establish a new server, the following steps are necessary:

1. Install Erlang (if not already present) and RabbitMQ on the server-to-be, following the installation instructions from the RabbitMQ website. Note in particular that, if using a non-administrative account, it will be necessary to copy the file .erlang.cookie from system32/config/systemprofile to the user’s home path.
2. Ensure that the RabbitMQ Windows service is running. It should launch automatically upon initial installation.
3. Ensure that access to port 5672 is permitted through the server’s firewall.
4. Using a command prompt, navigate to the sbin folder in RabbitMQ’s install location and set up a new user using the following command (sans double quotes): “rabbitmqctl.bat add_user ‘test’ ‘password’”. This will create a new user profile with username “test” and password “password” on the server – if a different username or password is desired, simply replace the relevant field in the command, while maintaining its surrounding quotation marks and escaping any necessary characters.
5. To enable the event monitor to pick up events from the server, also activate RabbitMQ’s “firehose” feature by running the following command (again, sans double quotes): “rabbitmqctl.bat trace_on”. Note that this command in particular must be run again every time the server is restarted.

To launch a Celery worker, ensure that tasks.py and worker_start.bat are located in the same directory, then run worker_start.bat from a command prompt. The worker can then be stopped by the ctrl-c keyboard shortcut or by closing its command prompt window. While running, unless this behavior has been modified, it will log all events to workerlog.txt including its startup messages, any tasks received or completed, and any errors encountered. Unless making modifications such as the proposal above regarding separating concurrency limits by queue, exactly one such worker should be launched on each machine which will be hosting NEMS runs, from a user account with all necessary permissions to complete those runs.

To launch a NEMS run through Celery, ensure that the program run_task.py is present in the scripts/setup/src/cel folder, or if it is to be relocated ensure that nems_setup.py has its import and shutil.copy statements modified accordingly. Additionally, ensure that the variable NEMSPYENV is set to an environment with the Celery package and all dependencies installed – this should be done both for the environment variable through a shell command and in the scedes file to be used. These prerequisites being met, simply execute runnems.bat from a command line as usual and make all appropriate selections. A command prompt window will be launched, per previous behavior. It is not recommended to close this command prompt window until the run has been completed.

To operate the run monitor, it is necessary to leave a single instance of nemseventmonitor.py running continuously, which will keep the output HTML file up to date as it receives new events from the RabbitMQ server. This instance can be located on any computer and account with access and permissions to modify files in the desired output directory, which is set to Z:/onl_tst2/sj currently. To launch it, execute eventmonitor_start.bat in a command prompt window.

It is recommended that the run monitor be launched immediately after executing the "rabbitmqctl.bat trace_on" command on the RabbitMQ server and before any runs are launched; any runs which started before launching the monitor may not be properly tracked and may generate junk output. If the run monitor is launched while previous HTML output files exist in the target directory, it will overwrite them, so be sure to back up any previous monitor output which needs preservation before launching a new instance. This behavior can also be used to clean up any junk output generated as a result of runs executed prior to the run monitor's launch; however, any runs which were started before launching the run monitor, and which are still running, may generate new junk output upon completion of subtasks and/or of the run itself.

The output of the run monitor will be two HTML files, one which discards old records after two days and one which discards them after a week. These files can be read using any web browser, and refreshing the page will update it for any changes have occurred since it was first opened. The run monitor checks for log file updates at an interval of once per minute, as well as automatically updating whenever a message is received from the RabbitMQ server indicating that a new task has started or has been completed. Clicking on column headers permits sorting in ascending or descending order by the value of that column, though any sorting will not be preserved through a page update.

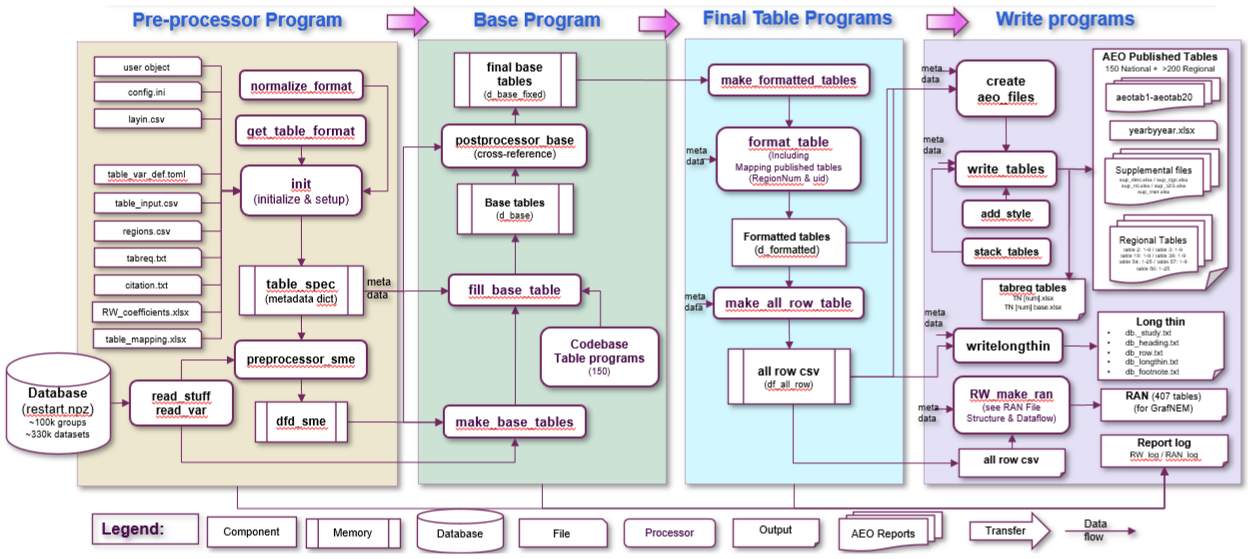
The NEMS Report Writer

The NEMS Report Writer, debuted in AEO2025, produces outputs from NEMS. It gathers data from the restart file (the NEMS common database) to generate various outputs, including Excel files for publication, Excel files for analysis, CSV files for data visualization, and other reports to support debugging and analysis. It receives data from the restart file (a binary database containing tables stored in npz format) and a set of standalone input files.
It is independently callable outside of a larger NEMS run, allowing for testing and modifications to tables without the need to rerun the entire NEMS system.

NEMS Report Writer Structure

The NEMS Report Writer is structured modularly. It sequentially runs a preprocessor that prepares the data, a base program that converts that data into tables, a final table program that formats those tables, and then has a series of write routines to convert the formatted tables into publication tables.

Figure 7: Report Writer system diagram



Running the Report Writer

The Report Writer is designed to run independently or to be callable inside of a large NEMS run. It requires essential information specific to the NEMS run, which is not available in the restart database or input files (such as the restart file location, Study_ID, etc.) for generating NEMS tables. This information must be provided to NEMS_RW at the time of initiation.

Two methods have been designed to convey the required information to the reporting platform:

1. Updating the User object by modifying the User object in RW_reporter_main.py (see the top right).
2. Updating the user.csv File by modifying the user.csv file located in the package's main folder (see the bottom right). This file should be used as an augment when running the program.

Data in the user.csv file will overwrite the User object if the Report Writer is kicked off with an argument like the following: *Python RW_reporter_main.py user.csv.*

Config.ini should be adjusted to set operational parameters, and tabreq.txt should be adjusted to indicate tables to print.

Preprocessor Program

The preprocessor processor program processes the inputs; both configuration files that tell the report writer what to do, as well as data files that are used to populate the final tables. These inputs include the following:

Table 4: Preprocessor input files

Filename	Brief description
user.csv	Configuration information that is used when the report writer is used outside the larger NEMS framework
tabreq.txt	Indicates which tables are printed
table_var_def.toml	Holds lists of required variables by table
table_mapping.xlsx	Holds mapping from layin tables to AEO published tables
table_input.csv	Specifies table ID, region name, and Table Program name
RW_coefficients.xls	Holds conversion ratios and constants
regions.csv	Holds regional information
layin.csv	Specifies the key elements of each row for each table
citations.txt	Citations and corresponding values used in publication
config.ini	This file consists of two sections: Debugging and Settings

Base Program and Table Programs

There are 150 table programs, one corresponding to each data table modelers use to publish and/or review specific data coming out from the model. For example, Table 1 titled “Total Energy Supply, Disposition, and Price Summary” has rows from various NEMS modules giving information about different fuels such as Natural Gas, Coal, Nuclear, Other Renewable Energy to give the reader an insight as to the data for the selected range of dates. Other tables break down individual module sections into smaller regional based data or fine details about all of the output from a module. To do this, NEMS Report Writer uses the layin file, the table_var_def.toml file, and the RW Tables/RW_fill_table_base_XXX.py (where XXX is the table number) to understand what rows the program should print out to output files. The Layin.csv file gives the layout and formatting for output, the table_var_def.toml file defines what variables will be used from the global data structure for the output, and the RW_fill_table_base_xxx.py file holds the calculations to compute to fill out the data. By using the make_base_tables in the base program, an unformatted table is generated first, holding in memory all of the associated pieces that would generate a table for review by a user.

Postprocessor

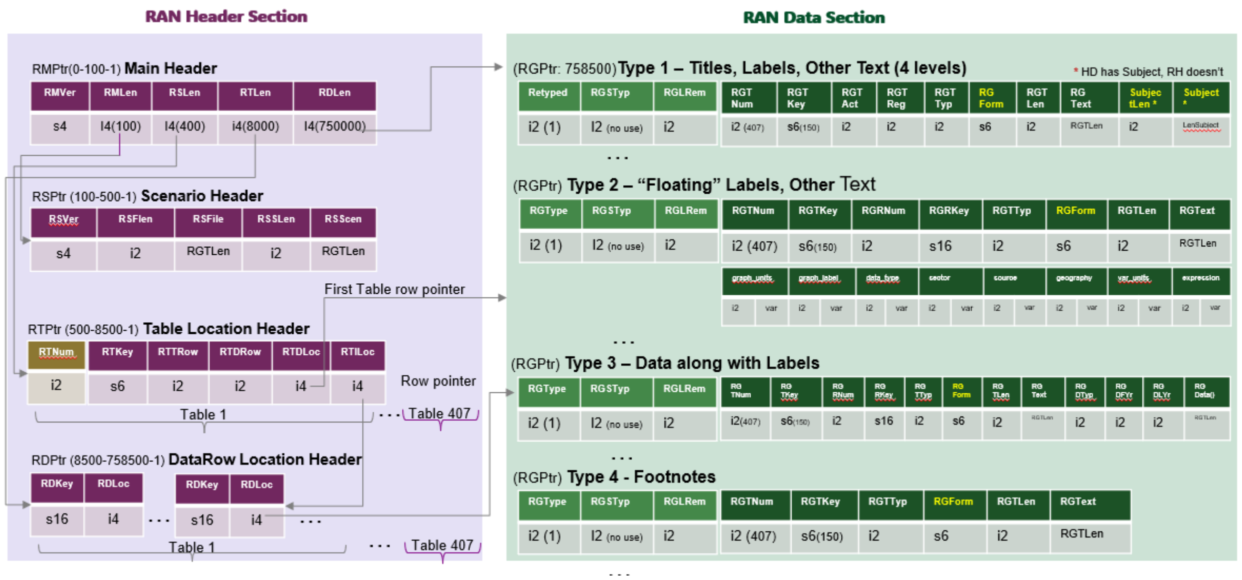
Some tables require data from other tables that are calculated and cannot be retrieved from the restart, resulting in difficulties completing calculations for certain data rows within the individual table programs.

To address these issues, a special component called postprocessor_base.py was designed. This component runs after the base table programs to “fill in” placeholders defined in those table program. Additionally, some calculations can be performed in batches within this component and then utilized by individual tables to improve performance.

RAN File Generator

RAN is a randomly accessible binary file used for visualizing NEMS projections in GrafNEM.
The RW_make_ran.py code was developed to generate the RAN file, which stores all NEMS tables) in binary format, based on the provided documentation and FTAB references. RW_make_ran.py is a component that can be run independently or integrated into the NEMS_RW platform (invoked by the main function of RW_reporter_main.py). It reads table data from the "all row csv.csv" file generated by the reporting platform and obtains table and row formatting information from layin.

Figure 8: RAN File Generator diagram


RAN Header Section
RMPtr(0-100-1) Main Header
RMVer | RMLen | RSLen | RTLen | RDLen
s4 | I4(100) | I4(400) | i4(8000) | I4(750000)
RSPtr (100-500-1) Scenario Header
RSVer | RSFlen | RSFile | RSSLen | RSScen
s4 | i2 | RGTLen | i2 | RGTLen
First Table row pointer
RTPtr (500-8500-1) Table Location Header
RTNum | RTKey | RTTRow | RTDRow | RTDLoc | RTILoc
i2 | s6 | i2 | i2 | i4 | i4
Row pointer
Table 1
. . .
Table 407
RDPtr (8500-758500-1) DataRow Location Header
RDKey | RDLoc
s16 | i4
. . .
RDKey | RDLoc
s16 | i4
. . .
Table 1
. . .
Table 407
RAN Data Section
(RGPtr: 758500) Type 1 – Titles, Labels, Other Text (4 levels)
* HD has Subject, RH doesn't
Retyped | RGSTyp | RGLRem | RGT Num | RGT Key | RGT Act | RGT Reg | RGT Typ | RG Form | RGT Len | RG Text | SubjectLen * | Subject *
i2 (1) | I2 (no use) | i2 | i2 (407) | s6(150) | i2 | i2 | i2 | s6 | i2 | RGTLen | i2 | LenSubject
. . .
(RGPtr) Type 2 – “Floating” Labels, Other Text
RGType | RGSTyp | RGLRem | RGTNum | RGTKey | RGRNum | RGRKey | RGTTyp | RGForm | RGTLen | RGText
i2 (1) | I2 (no use) | i2 | i2 (407) | s6(150) | i2 | s16 | i2 | s6 | i2 | RGTLen
graph_units | graph_label | data_type | sector | source | geography | var_units | expression
i2 | var | i2 | var | i2 | var | i2 | var | i2 | var | i2 | var | i2 | var | i2 | var
. . .
(RGPtr) Type 3 – Data along with Labels
RGType | RGSTyp | RGLRem | RG TNum | RG TKey | RG RNum | RG RKey | RG TTyp | RG Form | RG TLen | RG Text | RG DTyp | RG DFYr | RG DLYr | RG Data()
i2 (1) | I2 (no use) | i2 | i2(407) | s6(150) | i2 | s16 | i2 | s6 | i2 | RGTLen | i2 | i2 | i2 | RGTLen
. . .
(RGPtr) Type 4 - Footnotes
RGType | RGSTyp | RGLRem | RGTNum | RGTKey | RGTTyp | RGForm | RGTLen | RGText
i2 (1) | I2 (no use) | i2 | i2 (407) | s6(150) | i2 | s6 | i2 | RGTLen
. . .

The NEMS Validator

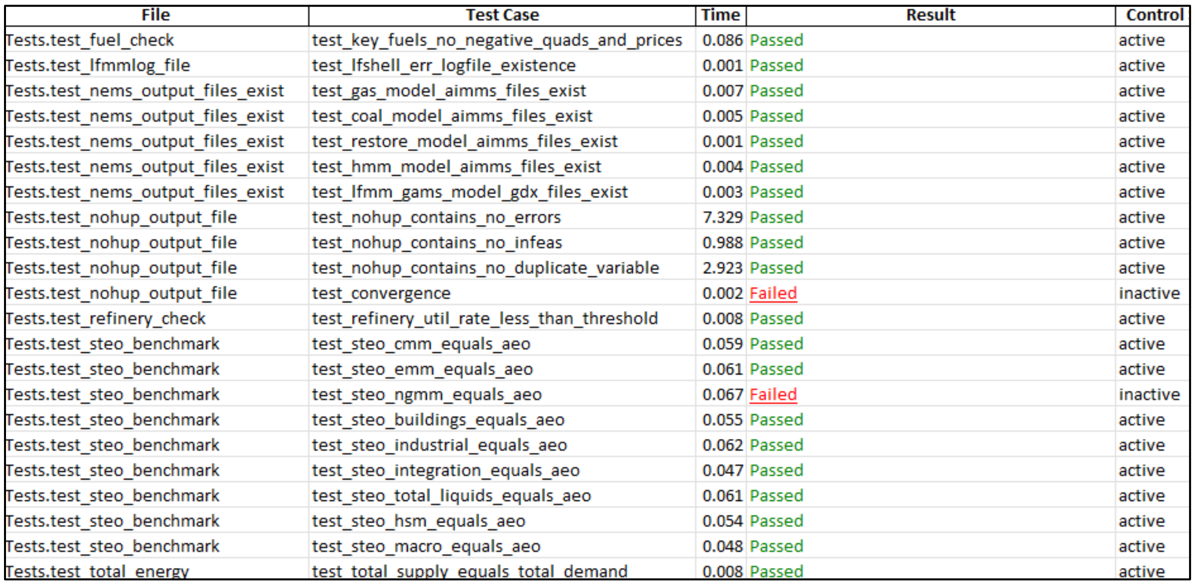
The NEMS Validator, a python program written using the pytest framework, tests NEMS results to see if they are appropriate for publication. After each NEMS run is completed, the validator runs checks that answer questions such as:

1. Are there error codes in any NEMS logs?
2. Is NEMS properly calibrated to STEO and SEDS?
3. Do select subtotals add to the appropriate totals?
4. Do expected output files exist?
5. Are energy prices and quantities all positive?
6. Does total energy supply equal total demand?
7. Did the model converge?

If all tests are successful the validator writes out, in a root directory of the run, a file named "Validator_pass.xlsx." Conversely, if any test fails, the validator names the files "Validator_fail.xslx."

Tests can be active, inactive, or off. Inactive tests run, but a failed results does not affect the file name being "fail" or "pass."

Figure 9: Sample Validator Tests



File	Test Case	Time	Result	Control
Tests.test_fuel_check	test_key_fuels_no_negative_quads_and_prices	0.086	Passed	active
Tests.test_lfmmlog_file	test_lfshell_err_logfile_existence	0.001	Passed	active
Tests.test_nems_output_files_exist	test_gas_model_aimms_files_exist	0.007	Passed	active
Tests.test_nems_output_files_exist	test_coal_model_aimms_files_exist	0.005	Passed	active
Tests.test_nems_output_files_exist	test_restore_model_aimms_files_exist	0.001	Passed	active
Tests.test_nems_output_files_exist	test_hmm_model_aimms_files_exist	0.004	Passed	active
Tests.test_nems_output_files_exist	test_lfmm_gams_model_gdx_files_exist	0.003	Passed	active
Tests.test_nohup_output_file	test_nohup_contains_no_errors	7.329	Passed	active
Tests.test_nohup_output_file	test_nohup_contains_no_infeas	0.988	Passed	active
Tests.test_nohup_output_file	test_nohup_contains_no_duplicate_variable	2.923	Passed	active
Tests.test_nohup_output_file	test_convergence	0.002	Failed	inactive
Tests.test_refinery_check	test_refinery_util_rate_less_than_threshold	0.008	Passed	active
Tests.test_steo_benchmark	test_steo_cmm_equals_aeo	0.059	Passed	active
Tests.test_steo_benchmark	test_steo_emm_equals_aeo	0.061	Passed	active
Tests.test_steo_benchmark	test_steo_ngmm_equals_aeo	0.067	Failed	inactive
Tests.test_steo_benchmark	test_steo_buildings_equals_aeo	0.055	Passed	active
Tests.test_steo_benchmark	test_steo_industrial_equals_aeo	0.062	Passed	active
Tests.test_steo_benchmark	test_steo_integration_equals_aeo	0.047	Passed	active
Tests.test_steo_benchmark	test_steo_total_liquids_equals_aeo	0.061	Passed	active
Tests.test_steo_benchmark	test_steo_hsm_equals_aeo	0.054	Passed	active
Tests.test_steo_benchmark	test_steo_macro_equals_aeo	0.048	Passed	active
Tests.test_total_energy	test_total_supply_equals_total_demand	0.008	Passed	active