Emissions Policy 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

EPM Documentation

EPM is the **E**missions **P**olicy, **M**odule of the National Energy Modeling System (NEMS). EPM and NEMS are developed by the U.S. Energy Information Adminstration. This site documents the inputs, formulation, and source code of the EPM.

EPM is written in Python. Documentation for the source code of the CCATS module can be found in the Model API Reference Section.

Introduction

EPM is the **E**missions **P**olicy **M**odule of the National Energy Modeling System (NEMS). The main purpose of the EPM is to handle calculations of energy-related carbon dioxide (CO_2) emissions at the U.S. economic sector and regional levels. These calculations are performed using energy consumption estimates (which vary by sector, region, and year) and applying applying appropriate sector-by-fuel emissions factors (which are established using the latest historical data and are static over the projection). In addition to providing projections of CO_2 emissions, the EPM is also responsible for implementing various CO_2 policy evaluation options. These options can be used to simulate proposed market-based approaches to meet national CO_2 emission objectives.

The CO_2 emissions estimates and policies modeled by the EPM focus specifically on energy-related CO_2 emissions. We define energy-related CO_2 emissions as those resulting from fossil fuel combustion, released during non-fuel use of energy products (such as industrial feedstocks), and released during energy production (such as CO_2 vented from geothermal wells). This distinction between energy- and non-energy CO_2 emissions categories in NEMS is discussed further in the Model Assumptions section.

Annual Model Updates

This edition of the Emissions Policy Module (EPM)-Model Documentation 2025 reflects changes made to the EPM since the publication of the 2023 Annual Energy Outlook. These changes include:

- Updates to carbon dioxide (CO_2) emissions factors
- Additional CO_2 emissions factors to represent new fuel usages for AEO2025
- Added representation of vented CO_2 emissions associated with natural gas processing
- Changes to the code base from Fortran to Python

Model Assumptions

Overview

The *Annual Energy Outlook 2025* (AEO2025) projects carbon dioxide (CO_2) emissions by fuel and by sector for three energy-related activities:

- Fossil fuel combustion
- Nonfuel use of fossil fuels (for example, in industrial activities such as manufacturing plastics)
- Naturally occurring CO_2 vented during energy consumption or production (for example, geothermal or natural gas processing)

For each activity, we estimate projected CO_2 emissions by multiplying associated energy consumption of each fuel by a CO_2 emission factor. Emissions factors reflect the amount of CO_2 emitted per unit of energy consumed and are expressed as millions of metric tons (MMmt) of CO_2 per quadrillion British thermal units (quads) of energy use.

To calculate CO_2 emissions factors, we start with CO_2 coefficients at full combustion for each fuel type. We adjust each coefficient by multiplying it with a combustion fraction between 0.0 and 1.0, arriving at an adjusted CO_2 emission factor for each fossil fuel. We assume all fuels are fully emissive when combusted (that is, a combustion fraction of 1.0). For nonfuel uses, the combustion fraction reflects our estimates of how much carbon remains in the product instead of being released into the atmosphere. We assume some nonfuel uses of fossil fuels capture all carbon inputs but other nonfuel uses emit some CO_2 during production. Emissions factors and combustion fractions for all fossil fuel categories are listed below.

Fossil fuel combustion

CO_2 emissions from fuel use vary based on the:

- Carbon content of the fossil fuel
- Fraction of the fuel combusted
- Amount of the fuel consumed

The chemical composition of most fossil fuels is relatively consistent over time, resulting in little to no change in their carbon factors over our AEO projections. However, some fuel categories have greater variability. For example, coal is reported as a single fuel type, but if the underlying coal ranks that make up the coal category change, the carbon factor can change over time.

For fuel uses of energy, we assume all of the carbon is oxidized, so the combustion fraction is equal to 1.0 (in keeping with international convention). Some products, such as petroleum coke, have both fuel and nonfuel uses, and we adjust the combustion fraction accordingly. Lubricants are not used for their energy value, but we assume that half of the lubricants consumed are combusted (therefore, emitted) and half are not.

Nonfuel use (Fuel-dependent processes)

CO_2 emitted during nonfuel energy use varies widely across energy products. For some products, such as asphalt and road oil, we assume that all CO_2 is captured during nonfuel uses. As a result, the adjusted CO_2 emissions factor is zero. For other fossil fuel inputs, such as those for petrochemical feedstocks, some CO_2 is emitted during production, and some carbon is stored in a final product (and not emitted into the atmosphere), reducing the fuel's CO_2 emissions relative to full combustion.

Biomass combustion

By convention, we assume biomass combustion results in net-zero CO_2 emissions. Specifically, we consider any CO_2 emitted by biogenic energy sources, such as biomass and alcohols, to be balanced by the CO_2 sequestration that occurred during biomass production.

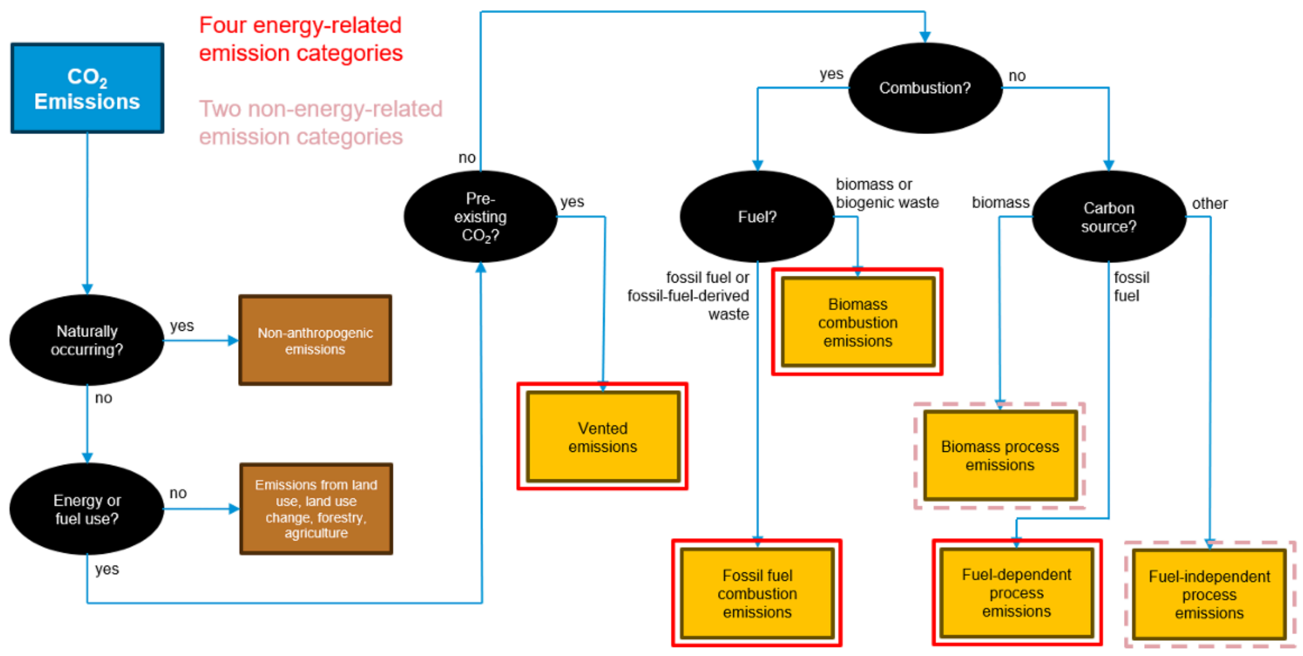
For fuels or fuel categories containing only biogenic fuels (such as woody biomass or biogenic municipal solid waste), CO_2 emissions are reported as zero. For fuels that contain both biogenic and non-biogenic components, such as ethanol blended with motor gasoline or biodiesel, biogenic components are excluded from emissions calculations. To illustrate the potential for these emissions in the absence of any offsetting sequestration-as might occur under related land-use changes, with CO_2 being sequestered in terrestrial carbon sinks-we calculate and report the CO_2 emissions from biogenic fuel use separately. However, these values are not included in total or sectoral emissions estimates.

Fuel-independent processes

Some industrial processes release CO_2 as a result of natural chemical processes, rather than through the fuel or nonfuel use of energy products. One example is CO_2 released from limestone during cement production. Although these emissions contribute to an overall national total, they are outside the scope of what we consider to be energy related. As such, we calculate and report these CO_2 emissions separately, but we do not include these values in our total or sectoral energy-related emissions estimates.

Reporting

Figure 2 clarifies how we distinguish energy and non-energy CO_2 emissions in our published AEO tables.


Four energy-related emission categories
Two non-energy-related emission categories
CO_2 Emissions
Naturally occurring?
yes
Non-anthropogenic emissions
no
Energy or fuel use?
no
Emissions from land use, land use change, forestry, agriculture
yes
Pre-existing CO_2?
no
yes
Vented emissions
Combustion?
yes
no
Fuel?
biomass or biogenic waste
Biomass combustion emissions
fossil fuel or fossil-fuel-derived waste
Fossil fuel combustion emissions
Carbon source?
biomass
Biomass process emissions
fossil fuel
Fuel-dependent process emissions
other
Fuel-independent process emissions

Source: U.S. Energy Information Administration

Figure 2. *Energy and non-energy emissions categories in NEMS*

The CO_2 emissions reported in AEO2025 Table 18 and Table 70 include all energy-related emissions from fossil fuel combustion, fuel-dependent processes, and venting. Table 18 groups these emissions by economic sector and fuel type, and Table 70 groups them by economic sector and end use. Table 71 reports CO_2 emissions by all categories shown in Figure 2, with subtotals for energy and non-energy CO_2 emissions.

Table 1. CO_2 emissions factors

million metric tons of CO_2 per quadrillion British thermal units

Fuel type	CO_2 coefficient at full combustion	Combustion fraction[a]	Adjusted emission factor
Petroleum			
Propane used as fuel	62.88	1.0	62.88
Propane used as feedstock	62.88	0.2	12.58
Ethane used as fuel	59.58	1.0	59.58
Ethane used as feedstock	59.58	0.2	11.92
Butane used as fuel	64.75	1.0	64.75
Butane used as feedstock	64.75	0.2	12.95
Isobutane used as fuel	64.94	1.0	64.94
Isobutane used as feedstock	64.94	0.2	12.99
Natural gasoline (pentanes plus) used as fuel	66.88	1.0	66.88
Natural gasoline (pentanes plus) used as feedstock	66.88	0.2	13.38
Motor gasoline (not including ethanol)	70.66	1.0	70.66
Jet fuel	72.23	1.0	72.23
Distillate fuel (not including biodiesel)	74.14	1.0	74.14
Residual fuel	75.09	1.0	75.09
Asphalt and road oil	75.35	0.0	0.0
Lubricants	74.07	0.5	37.03
Petrochemical feedstocks	69.26	0.308	21.31
Kerosene	73.19	1.0	73.19
Petroleum coke (industrial)	102.12	0.956	97.59
Petroleum coke (electric power)	102.12	1.0	102.12
Petroleum still gas	66.73	1.0	66.73
Other industrial[b]	30.39	1.0	30.39
Coal			
Residential and commercial	95.99	1.0	95.99
Metallurgical	93.90	1.0	93.90
Coke	113.67	1.0	113.67
Industrial other[c]	95.70	1.0	95.70
Electric power[d]	95.81	1.0	95.81
Natural gas			
Used as fuel	52.91	1.0	52.91
Used as hydrogen production feedstock	52.91	1.0	52.91
Used as other industrial feedstock	52.91	0.366	19.37
Biogenic energy sources[e]			
Woody biomass	93.81	1.0	93.81
Biogenic waste	90.64	1.0	90.64
Biofuels heat and coproducts	93.81	1.0	93.81
Ethanol	68.42	1.0	68.42
Biodiesel	72.73	1.0	72.73
Renewable diesel and gasoline	73.15	1.0	73.15
Renewable natural gas	52.91	1.0	52.91
Biobutanol	70.58	1.0	70.58
Other biomass liquids	73.15	1.0	73.15

Data source: U.S. Energy Information Administration, *Annual Energy Outlook 2025,* National Energy Modeling System run: ref2025.d032025a, and Appendix tables A-20, A-32, and A-226, U.S. Environmental Protection Agency (EPA), *Inventory of U.S. Greenhouse Gas Emissions and Sinks: 1990–2022*

Note: Emissions coefficients from EPA are converted from units of carbon to CO_2 by multiplying by a factor of (44/12).[a] For feedstocks, the combustion fraction includes fuel-dependent process emissions as well as inputs that might be combusted onsite.

[b] *Other industrial petroleum* includes industrial lubricants, special naphtha (solvents), waxes, and miscellaneous products such as sulfur.

[c] *Industrial other coal* is for process heat, and qualitatively differs from coal used for steel production (metallurgical coal).

[d] The National Energy Modeling System specifies emission factors for coal used for electric power generation by coal supply region and types of coal, so the average CO_2 content for coal varies throughout the projection period. The electric power value of 95.81 shown here illustrates a typical coal-fired emission factor.

[e] We include biogenic sources for informational purposes, but we do not count them in total energy-related CO_2 emissions.

Inputs and Methods

The EPM is called at the end of each NEMS iteration, after all other modules have been called. The module uses energy consumption projections from other NEMS modules as well as exogenous carbon dioxide (CO_2) emissions factors to create projections of energy-related CO2 emissions. In addition to emissions calculations, if CO_2 policy cases are enabled, some form of energy price adjustment is calculated to account for the CO_2 tax, or permit fee, for the next iteration. The CO_2 fee is either fixed (for a straight CO_2 tax) or is varied in each NEMS iteration until a CO_2 goal is met (for the permit auction and permit market options).

The fee on CO_2 emissions is modeled as an adjustment on the end-use price of the fuel. Two sets of end-use price variables are maintained in NEMS: an unadjusted set of prices without any CO_2 fee added, and an adjusted set of prices that includes the CO_2 fee. The unadjusted prices are those determined by the NEMS supply and conversion modules. The adjusted prices, with the CO_2 fee included, are the price variables used by the demand and conversion modules purchasing the fuel. In the Integrating Module, after each module is executed, the adjusted fuel prices are recalculated based on the current unadjusted fuel price and CO_2 fee.

Module inputs and outputs

The input data for the EPM comes primarily from other modules of NEMS. Exogenous data include the policy options to be implemented and the CO_2 emission factors. If a CO_2 tax scenario is to be implemented, the tax rate must also be specified. Alternatively, a CO_2 goal may be specified, and the CO_2 tax to meet that goal will be set in the EPM once per iteration of the NEMS solution algorithm. Output from the EPM consists of the volumes of CO_2 emissions by fuel and economic sector, adjustments to the end-use prices of fuels consumed by the demand and conversion modules, and revenue accrued based on these adjustments. The adjustments are additions to prices in 1987 dollars per million Btu. Revenue, in billions of 1987 dollars, from the CO_2 penalty is also calculated and can be used by the Macroeconomic Activity Module or for offline analysis of macroeconomic feedbacks.

Total energy-related CO_2 emissions from both combustion and non-combustion sources are calculated in the EPM from information in several NEMS common blocks. In many cases, CO_2 emissions are calculated using QBLK, which contains the projected quantities of end-use fuels consumed, and EMEBLK, which contains the CO_2 emissions factors to convert energy consumption into CO_2 emissions. Some additional common blocks are called to address more specific emissions estimates:

- QMORE, INDOUT, and BIFURC - are called to calculate non-combustion or feedstock use of some fossil fuels
- PMMRPT - to remove biodiesel and ethanol from transportation petroleum emissions
- UEFDOUT - for electric power sector natural gas consumption
- COALEMM - for projected electric power sector coal consumption by sulfur dioxide classification category and associated CO2 for each category
- TRANREP - to account for transportation sector electric power consumption
- HMMBLK - for heat and power, as well as feedstock, use of natural gas in hydrogen production
- QSBLK - for CO2 calculations specific to California
- CCATSDAT - for CO2 capture and storage volumes
- OGSMOUT - for vented CO2 emissions released during natural gas processing
- COGEN - for vented CO2 emissions released during geothermal electricity generation
- WRENEW - for consumption of biogenic and non-biogenic municipal solid waste for electricity generation

The resulting CO_2 emissions estimates are stored in the GHGREP common block. The inputs and outputs associated with the EPM CO_2 price policies are also stored in the NEMS global data structure. As input, these common blocks contain the NEMS end-use fuel prices:

- MPBLK
- NGTDMOUT
- COALPRC
- PMORE
- EUSPRC
- PONROAD

These prices, established in the NEMS supply and conversion modules, are the EPM input prices. As output, the EPM projects a dollar-per-Btu adjustment to each product-sector price to reflect any CO_2 tax or allowance fee. The adjustment is added to the NEMS end-use fuel prices, and they are stored in a parallel set of price common blocks:

- AMPBLK
- ANGTDM
- ACOALPRC
- APMORE
- AEUSPRC
- APONROAD

When no CO_2 policy options are in effect, the adjusted price common blocks match the unadjusted price common blocks from the supply modules. The energy price adjustments, equal to the difference between the two sets of prices, are stored in the EMABLK common block. If nonzero, these price adjustments are used as starting values when either of the CO_2 goal options (auction or permit market) are in effect. Several policy options result in revenue from the CO_2 penalty flowing to the government. This revenue is furnished to the Macroeconomic Activity Module through the EMISSION common block.

Module algorithm

The EPM is executed once per iteration to determine total CO_2 emissions produced, the revenue created by any tax or permit fees for CO_2 emissions and, depending upon the scenario, the level of offsets produced. For CO_2 emission policy options, a heuristic algorithm (Regula Falsi) sets a new CO_2 fee to bring the CO_2 emissions closer to the selected policy CO_2 goal.

The general flow of EPM, including relevant function calls and variable names, is as follows: * First year, first iteration processing

- Integrating module nems_flow.py executes run_epm.py in "read" mode, which runs the 'epm_read' function from epm_read.py
- 'epm_read' reads the policy switches in the control file, epmcntl.toml, and parses emissions-related data from several other EPM input files:
    - Emissions policy options are read in through four binary variables ('tax_flag', 'permit_flag', 'market_flag', 'offset_flag).
        - For emissions tax policies, additional switches are available to apply the tax to specific sectors ('elec_flag', 'tran_flag', 'resd_flag', 'comm_flag')
        - For emissions market policies, parameters can be set to specify program elements ('bank_flag', 'bank_startyr', 'bank_endyr', 'bank_end_balance')
    - Historical CO_2 emissions data are read in through epm_history.csv
    - Data for emissions policy cases is read in from epm_tax_or_cap.csv
    - A mapping of Coal Market Module regions are mapped to Census regions through epm_coal_regions.toml
    - Yearly CO_2 emissions factors for each fuel and sector are read in through epm_carbon_factors.tsv
    - Mercury emission classes and caps (for use in the Electricity and Coal Market Modules) are read from epm_mercury_classes.csv and epm_mercury_caps.csv. Additional mercury parameters, including control technologies and mercury emissions rates are read through epm_mercury_parameters.toml
    - Data and parameters pertaining to the California AB-32 cap and trade program are read in through epm_ab32_data.csv and epm_ab32_parameters.toml
    - Exogenous emissions baselines for other (non-CO_2) gases and offset assumptions (marginal abatement cost tables) are read from ghgoffx.xlsx
    - Emission allowance auction shares are read from epm_restart.py

- Nems_flow.py executes run_epm.py in “main” mode, which calls the core ‘epm’ function from epm_core.py
- ‘epm’ then calls on several functions from other supporting Python files to perform various EPM operations
    - The ‘sum_emissions’ function (from epm_sum_emissions.py) adds up CO_2 emissions across all sectors of the economy, shares emissions by region and electrical power usage, handle historical benchmarking/overwrites, and report the totals.
        - For the market permit system with offset policy option, the ‘sum_emissions’ function calls the ‘oghg’ function (from epm_other_ghg.py) to determine what level of emission offsets is available to raise the CO_2 cap, given the current CO_2 tax
    - The ‘accntrev’ function (from epm_revenue.py) calculates revenues from CO_2 tax or CO_2 permit fees from emissions policy cases
    - For emissions policy options including a market permit system, the ‘initrev’ function (from epm_revenue.py) allocates revenue to end-use sectors based on initial sector shares of CO_2 emissions
    - For emissions policy options including a permit auction or market, the ‘regfalsi’ function (epm_regula_falsi.py) calculates new CO_2 taxes to reduce the absolute difference between the CO_2 emissions and an established CO_2 goal
    - For emissions policy options including a CO2 tax, the ‘price_adjust’ function (from epm_adjustments.py) adjusts energy prices for end-use fuels are adjusted by multiplying the CO_2 tax by the fuels’ emission factor
- After each NEMS module is called, prices are recalculated to include a tax by adding the tax price to the prices projected by the supply modules (though the ‘copy_adjusted’ function from pyfiler1.py in the NEMS main module).

Emissions policy options

In addition to providing estimates of energy-related CO_2 emissions, one of the EPM’s primary functions is to model hypothetical emissions policies. The EPM can model five different policy scenarios in NEMS. Each of these policy cases can be turned on or off by adjusting their associated binary ‘flag’ variables in the EPM control parameters file (epmcntl.toml). Descriptions of each policy scenario are provided below.

Carbon dioxide tax

A tax per kilogram of carbon for fossil fuels is converted to a dollar-per-Btu tax and applied to the prices for each fuel consumed in each sector covered by the tax, based on the CO_2 emission factor for that fuel and sector. The tax can be input in either nominal or real dollars, and a different tax may be set for each projection year. Fossil fuel prices are adjusted to include the tax. Variables represent the unadjusted prices that are filled by the supply modules. The size of the adjustment or tax that the EPM fills yields the adjusted prices. These adjusted prices are used by the demand and conversion sector modules to simulate the effect that the tax has on CO_2 emissions levels. Projected revenue from the tax is passed to the Macroeconomic Activity Module, where allocation of such revenue (for example, a deficit-neutral return to consumers) depends on a user-specified option setting. Generally, large changes in government revenue would require additional offline analysis to assess macroeconomic feedbacks.

Permit auctions

An auction to distribute emissions permits is simulated. The total number of permits sold corresponds to the total CO_2 emission goal that is set by the user. A different goal may be set for each projection year. Essentially, this option determines the permit fee necessary to achieve the CO_2 goal by clearing the auction market. The permit fee is treated as a CO_2 emissions tax and used as an adjustment to the fossil fuel prices. A new auction price is set at the end of each NEMS iteration (where one iteration in the solution algorithm refers to a single execution pass through all NEMS modules for a single projection year) until the emissions reach the goal. The permit auction is assumed to operate with no initial allocation of emission permits. Similar to the CO_2 tax option, revenue from the auction is passed to the Macroeconomic Activity Module where its effect may require additional analysis.

Market for permits

A market for tradable carbon dioxide emission permits is simulated with the assumption that an initial distribution of marketable permits to emission sources takes place. The permits are transferable. Depending on a user-specified model option, the permits may be treated as bankable across years. As with the CO_2 tax and auction options, the full market price of the permits is added to energy prices on a dollar-per-Btu basis. The system of marketable permits is implemented in the same way as in the permit auction, except the calculation of revenues from permit sales. Similar treatment is warranted because the marginal cost of a free permit is equivalent to one purchased at auction, given the opportunity cost of holding the distributed permit.

In an open, competitive permit market, the permit will tend to be priced at the marginal cost of reducing CO_2 emissions, regardless of the initial distribution of permits. If permits are purchased by suppliers and passed through to the fuel price, the marginal cost of the CO_2 emissions by a particular sector in a region will be reflected in the individual end-use fuel cost for that sector. The evaluation of the initial distribution of permits depends on the sector.

For those sectors in which the product prices are based on marginal cost, as in the Liquid Fuels Market Module, the value of the initial distribution of permits may be ignored; it does not affect the price of products. However, in the regulated electricity sector, where the average cost is used to determine price, the revenue attributed to the free use or sales of the initially distributed permits would possibly be passed through to the consumers. The value of the initial distribution of permits is calculated, but it is not used for electricity pricing. Instead, the full cost of the permits, as though there were no initial distribution, is reflected in the projected electricity price.

As with the auction, a new permit fee is set at the end of each NEMS iteration (where one iteration in the solution algorithm refers to a single execution pass through all NEMS modules for a single projection year). The fee is adjusted up or down in response to the total CO_2 emissions obtained. The price of an allowance is adjusted until the total carbon dioxide produced is within a tolerance of the goal for that year.

Market for permits with emission offsets

The offset option allows the goal on tradable emissions permits to increase through a user-specified supply of offsets, expressed as marginal abatement cost tables for other (non-CO_2) gases. This option can be used to analyze a greenhouse gas emission reduction policy that credits reductions in emissions from non-covered sources, reforestation, or purchases of emission reductions credits from abroad. Purchases of offsets, in millions of metric tons available at the given allowance price, are added to the CO_2 goal. Although some test values for offsets are available, any formal use of this option would require additional research to arrive at appropriate assumptions. The specification of offset supply curves, or marginal abatement cost tables, along with exogenous projections of greenhouse gases other than energy-related CO_2, are made through the ghgoffx.xlsx input file.

Early-compliance emissions allowance banking with smooth carbon fee growth

A cap and trade with banking is implemented by finding the starting carbon price, and then escalating the price at a fixed rate that clears the bank over the compliance period. The bank is determined as the sum of the emissions goal minus projected emissions over the relevant period. The starting price is guessed based on results of prior NEMS cycles. If no data from prior NEMS cycles are available to construct price estimates, then estimates are created from scratch The projected prices are set in the start year based on the guess, then the case is run as a carbon fee case.

The option to allow banking of emissions permits can also be combined with either the permit auction or permit market by adjusting the 'bank_flag' variable in the EPM control file (epmcntl.toml). The details of the banking policy are controlled by additional parameters beginning with the "bank" prefix in the control file (e.g., 'bank_startyr' and 'bank_endyr').

epm_adjustments module

Policy functions for applying price adjustments to the end-use fuels.

This file implements the *etax_adjust* and *price_adjust* functions, which calculate price adjustments for energy taxes and carbon prices, respectively.

`epm_adjustments.etax_adjust`(*restart: Restart*)→ None [source]

Calculate price adjustments (energy taxes) to apply to end-use fuels.

This function is called by the *epm* function if the energy tag flag is set in the EPMCNTL file. The bulk of the function assigns energy taxes to the appropriate price adjustment variables defined in the EMABLK common block. In each case, we are finding the price that the demand sectors respond to and adjusting that price based on an energy tax. This is implemented by comparing prices from the previous two NEMS cycles.

This is closely related to the *price_adjust* function, which carries out similar calculations for the case of carbon prices.

Parameters: **restart** (*Restart*) – The currently loaded restart file data.

See also

`epm`

Core EPM routine that calls this function to compute adjustments.

`price_adjust`

Similar function for the case of a carbon price.

`epm_adjustments.price_adjust`(*restart: Restart, scedes: Scedes*)→ None [source]

Calculate price adjustments (carbon content) to apply to end-use fuels.

Called by the *epm* function if the tag flag, market flag, or permit flag is set in the EPMCNTL file. The bulk of this function assigns carbon prices to the appropriate price adjustment variables defined in the EMABLK common block. In each case, we are finding the price that the demand sectors respond to and adjusting the price based on the carbon content of the fuel and the tax. This is implemented by comparing prices from the previous two NEMS cycles.

This is closely related to the *etax_adjust* function, which carries out similar calculations for the case of an energy tax.

Parameters:

- **restart** (*Restart*) – The currently loaded restart file data.
- **scedes** (*Scedes*) – The current scedes file dict.

See also

`epm`

Core EPM routine that calls this function to compute adjustments.

`etax_adjust`

Similar function for the case of an energy tax.

epm_common module

Constant values and general utility functions for EPM.

`epm_common.BASE_YR`: *Final[int] = 34*

Legacy *base_yr* constant for use in the *sum_emissions*, *oghg*, and *accntrev* functions.

It is the NEMS year index for the last year of history overwrites. When reading history in EPMDATA, 2023 is currently the last data year.

See also

`EPM_READ_HIST`

Same value for the *epm_read* function.

`epm_common.EPM_READ_HIST`: *Final[int] = 34*

Legacy *hist* constant for use in the *epm_read* function.

It is the number of years of emissions factor data, and indicates that we should read the historical CO2 data through NEMS year 34 (2023).

See also

`BASE_YR`

Same value for several other functions in EPM.

`epm_common.REGFALSI_EPM_FACTOR`: *Final[float] = 3.0*

Legacy *epm_factor* constant used by the *regfalsi* function.

This is a mutliplicative factor to branch outward by when trying to get the bracketing points.

See also

`REGFALSI_TOL`

Tolerance for the *regfalsi* function.

`epm_common.REGFALSI_TOL`*: Final[float]* *= 1.0*

Legacy *tol* constant used by the *regfalsi* function.

This is our tolerance on reaching the pollution goal, so that the function knows when it has found the root to the desired level of accuracy.

See also

`REGFALSI_EPM_FACTOR`

Bracketing factor for the *regfalsi* function.

`epm_common.REGFALSIBANK_CTB`*: Final[float]* *= 0.0*

Legacy *ctb* constant used by the *regfalsibank* function.

In the rebracketing request logic, if the last two emissions totals were bigger than this value, adjust the allowance price by slope rather than brackets.

See also

`REGFALSIBANK_EPM_FACTOR`

Bracketing factor for the *regfalsibank* function.

`REGFALSIBANK_TOL`

Tolerance for the *regfalsibank* function.

`epm_common.REGFALSIBANK_EPM_FACTOR`*: Final[float]* *= 0.01*

Legacy *epm_factor* constant used by the *regfalsibank* function.

This is a mutliplicative factor to branch outward by when trying to get the bracketing points.

See also

`REGFALSIBANK_CTB`

Adjustment threshold for the *regfalsibank* function.

`REGFALSIBANK_TOL`

Tolerance for the *regfalsibank* function.

`epm_common.REGFALSIBANK_TOL`*: Final[float]* *= 5.0*

Legacy *tol* constant used by the *regfalsibank* function.

This is our tolerance on reaching the long-term banking goal, so that the function knows when it has found the root to the desired level of accuracy.

See also

`REGFALSIBANK_CTB`

Adjustment threshold for the *regfalsibank* function.

`REGFALSIBANK_EPM_FACTOR`

Bracketing factor for the *regfalsibank* function.

`epm_common.set_log_file_path`(*file_path: Path*)→ None [source]

Set the path where EPM should save its EPMOUT log file.

An empty file will be created with the specified path, overwriting the file if it already exists.

Parameters: **file_path** (*Path*) – The log file path to use.

See also

`set_log_verbosity`

Set the logging verbosity level.

`log_it`

Write a message to the log file.

`epm_common.set_log_verbosity`(*verbosity: int*)→ None [source]

Set the verbosity level EPM should use when logging to the EPMOUT file.

The verbosity level corresponds to the *dbugepm* parameter in the EPM control file. It determines which messages (if any) get written to the EPMOUT log file.

Parameters: **verbosity** (*int*) – The desired verbosity level. There are three settings: 0 turns logging off, 1 turns logging on, and 2 enables additional verbose logging.

See also

`set_log_file_path`

Set the path to the log file.

`log_it`

Write a message to the log file.

`epm_common.log_it`(**message_parts: Any, sep: str = '\n', verbose: bool = False*)→ None [source]

Write a timestamped log message to the EPMOUT log file.

The provided parts of the message are converted to strings and then joined together with a separator before being written to the log file. Messages will only be written if logging is switched on, and messages flagged as "verbose" will only be written if the verbosity level is set to maximum.

Before ever calling this function, ensure that both the log file path and the logging verbosity have been set using their associated functions.

Parameters:
- **message_parts** (*Any*) – Zero or more arguments that will be joined together to form the log message. These objects should be convertable to strings.
- **sep** (*str, optional*) – The joining string to use between the message parts. Newlines are used by default.
- **verbose** (*bool, optional*) – If set to True, this message will be flagged as "verbose" and not written unless the logging verbosity level is set to maximum. Defaults to False.

Raises: **RuntimeError** – If this function is called before setting the log file path and verbosity level.

See also

`set_log_file_path`

Set the path to the log file.

`set_log_verbosity`

Set the logging verbosity level.

`epm_common.print_it`(*cycle: int, message: str*)→ None [source]

A standardized way to print an important message to stdout.

In integrated runs, this will be captured by the nohup.out file.

Parameters:
- **cycle** (*int*) – The current NEMS cycle number, also known as *curirun*.

- **message** (*str*) – The message to be printed.

epm_common.discover_run_configuration()→ tuple[pathlib.Path, bool] [source]

Infer the location of the EPM code and whether it is running integrated.

This implementation uses the expected directory structures for integrated and standalone EPM runs. It looks for a file named *run_epm.py* and returns the path to the directory containing that file. Depending on where the file was found, we can determine whether EPM is running integrated or standalone.

Returns:

- *Path* – Full path to the directory containing the EPM code.
- *bool* – Whether EPM is running integrated (True) or standalone (False).

Raises: **RuntimeError** – If the function is unable to definitively classify the EPM run configuration.

epm_common.running_integrated()→ bool [source]

Determine whether EPM is running integrated with the rest of NEMS.

This is equivalent to calling the *discover_run_configuration* function and discarding the path it returns.

Returns: True if EPM is running integrated with NEMS and False if EPM is running standalone.

Return type: bool

epm_common.get_epm_path()→ Path [source]

Get the path to the main EPM code directory for this run.

This is equivalent to calling the *discover_run_configuration* function and discarding the boolean flag it returns.

Returns: Full path to the main EPM code directory.

Return type: Path

epm_common.get_input_path()→ Path [source]

Get the path to the EPM input directory for this run.

Returns: Full path to the EPM input directory.

Return type: Path

epm_common.get_output_path()→ Path [source]

Get the path to the directory where EPM should save its output files.

Returns:	Full path to the EPM output directory.
Return type:	Path

`epm_common.incorporate_cycle_number`(*file_path: Path, cycle: int*)→ Path [source]

Change the given file path to include the current NEMS cycle number.

The cycle number is inserted with a '.' immediately before the file extension, so a file path of *C:/folder/example.txt* would be returned as *C:/folder/example.3.txt* for the third cycle.

Parameters:

- **file_path** (*Path*) – Original file path without cycle number incorporated.
- **cycle** (*int*) – The current NEMS cycle number, also known as *curirun*.

Returns: The new path including the cycle number.

Return type: Path

epm_core module

Core routines for the NEMS Emissions Policy Module (EPM).

The module is called from NEMS main as long as RUNEPM=1 is set in the scedes.

The primary role of this module is to add up carbon emissions from each sector of the U.S. economy in proportion to fuel consumption. That task is handled by *sum_emissions*, which is called from within the main *epm* function.

When required, EPM also implements carbon policy options – the first three policy options listed below are mutually exclusive (one at a time), the fourth option is a variation on the market policy, and the fifth option is a modifier that can be used with either the carbon cap or market policy:

1. Carbon Tax:

    A nominal or real $ tax per kilogram of carbon for fossil fuels. It is converted to a $/btu tax for each fuel/sector based on its carbon content. Revenue from the tax is passed to the macroeconomic module. There, treatment of such revenue (e.g., reducing the deficit or reducing other taxes, etc.) depends on options in effect. Generally, large changes in gov't revenue would require additional offline analysis to asses macroeconomic feedbacks.

    This policy is triggered by setting *tax_flag* in EPMCNTL and uses values from the section titled "EPM Carbon Tax or Carbon Cap Data" in EPMDATA. You may also want to set INTLFDBK=1, MACTAX=4, INDBMOVR=1, and RFHISTN=$NEMS/…/rfhist_nref.txt in the scedes.

2. Auction of Permits:

A carbon goal is specified by the user. The goal or cap is achieved by requiring an allowance to emit carbon. An auction clearing price is determined, through the NEMS iterative process, that will clear the auction market. Essentially, this option determines the carbon tax (or allowance fee) necessary to achieve the total carbon cap. The auction to price and allocate emissions allowances is assumed to operate with no initial allocation of allowances. As in option 1, Carbon Tax, revenue from the auction is passed to the macroeconomic module, where its effect may require additional analysis.

This policy is triggered by setting *auction_flag* in EPMCNTL and uses values from the section titled “EPM Carbon Tax or Carbon Cap Data” in EPMDATA. You may also want to set INTLFDBK=1, MACTAX=4, INDBMOVR=1, and RFHISTN=$NEMS/…/rfhist_nref.txt in the scedes.

3. Market for Permits:

   Same as option 2, Auction of Permits, but permits are transferable within the country (though not bankable). An initial distribution of emissions permits, equal to base year emissions, is assumed to take place. As a result, the revenue from the sale of allowances is assumed to be redistributed back to the individual sectors. For regulated electric utilities, the initial revenue from the allowance distribution would be considered independent of the cost (or opportunity cost) of purchasing the permits. The amount of this initial revenue or subsidy is calculated, but no treatment of it is performed for pricing purposes. The full cost of the permits, however, does feed through to the electricity price. Therefore, the effect on the electricity price is probably overstated (unless marginal cost pricing is assumed).

   This policy is triggered by setting *market_flag* in EPMCNTL and uses values from the section titled “EPM Carbon Tax or Carbon Cap Data” in EPMDATA. You may also want to set INTLFDBK=1, MACTAX=4, INDBMOVR=1, and RFHISTN=$NEMS/…/rfhist_nref.txt in the scedes.

4. Market for Permits with Emissions Offsets:

   Same as option 3, Market for Permits, but allows for the total cap on emission allowances to increase through a supply of offsets. The amount of offsets for reforestation (increasing carbon sequestration) and coal bed methane capture are specified at various permit prices; the higher the price, the greater the assumed offsets (i.e., a supply curve of exogenous emission allowance offsets). The offset (in tons) available at a given allowance price is added to the carbon goal.

   This policy is triggered in the same manner as option 3, but additionally requires setting *offset_flag* in EPMCNTL.

5. Early-Compliance Banking w/ Cap-and-Trade and Smooth Carbon Fee Growth:

A cap and trade with banking is implemented by finding the starting carbon price, escalated at a fixed rate, that clears the bank over the compliance period. The bank is determined as the sum of cap minus emissions over the relevant period. The starting price is guessed based on results of prior NEMS cycles. The projected prices are set in the start year based on the guess, then the case is run as a carbon fee case.

The banking policy can be combined with either option 2 or option 3, and is triggered by setting *bank_flag* in EPMCNTL. The details of the banking policy are controlled by additional parameters beginning with the *bank_* prefix in EPMCNTL (e.g., *bank_startyr* and *bank_endyr*).

`epm_core.epm`(*restart: Restart*, *scedes: Scedes*, *variables: Variables*)→ None [source]

Top-level routine for running the Emissions Policy Module (EPM).

Calling this function runs the core EPM code, but does not handle the reading of EPM's input files. See the module-level documentation for details on the Emissions Policy Module as a whole.

Parameters:
- **restart** (*Restart*) – The currently loaded restart file data.
- **scedes** (*Scedes*) – The current scedes file dict.
- **variables** (*Variables*) – The active intermediate variables for EPM.

See also

`epm_read`

Read the EPM input files and open the EPM output file.

`sum_emissions`

Called by this function to add up U.S. carbon emissions.

`epm_addoff`

Called by this function to account for carbon offsets.

`regfalsibank`

Called by this function to compute a starting carbon price.

`accntrev`

Called by this function to calculate emissions policy revenue.

`initrev`

Called by this function to compute value of permit allocation.

`regfalsi`

Called by this function to solve for a policy carbon price.

`etax_adjust`

Called by this function to calculate energy tax adjustments.

`price_adjust`

Called by this function to compute carbon price adjustments.

epm_fortran module

Stand-ins replacing Fortran functionality for the new Python EPM.

Everything in this file should eventually be made obsolete and deleted. These utilities exist only to simplify the process of translating Fortran code into Python. As the new Python code matures and is refactored, uses of these utilities should be replaced with more Pythonic solutions.

`epm_fortran.readrngxlsx`(*excel_file: BinaryIO, sheet_name: str*)→ DataFrame [source]

Provide a substitute for *subroutine ReadRngXLSX* defined in nemswk1.f.

The Fortran subroutine of this name defined in $NEMS/source/nemswk1.f reads all “defined ranges” from one worksheet of an Excel spreadsheet into memory for later retrieval with the *getrngr* and *getrngi* subroutines.

For this Python version, just read the entire Excel worksheet into a pandas DataFrame and return it. That DataFrame will then be passed into Python versions of *getrngr* and *getrngi* to extract the numeric data.

Parameters:
- **excel_file** (*BinaryIO*) – A file object reading in binary mode from the Excel file.
- **sheet_name** (*str*) – The name of the worksheet to read.

Returns: Unprocessed contents of the Excel worksheet.

Return type: pd.DataFrame

See also

`getrngr`

Used to extract float arrays from this function’s return value.

`getrngi`

Used to extract int arrays from this function’s return value.

`epm_fortran.getrngr`(*df: DataFrame, name: str, n_rows: Literal[1], n_cols: Literal[1]*)→ float64 [source]

`epm_fortran.getrngr`(*df: DataFrame, name: str, n_rows: int, n_cols: int*)→ ndarray[Any, dtype[float64]]

Provide a substitute for *subroutine getrngr* defined in nemswk1.f.

The Fortran subroutine retrieves arrays of reals from an Excel worksheet after *readrngxlsx* has already been called to read the file.

This Python version just searches through the DataFrame returned by *readrngxlsx* to find and process the requested range of data. It is returned as an array of floats with the appropriate number of dimensions.

Parameters:
- **df** (*pd.DataFrame*) – An Excel worksheet DataFrame returned by *readrngxlsx*.
- **name** (*str*) – The name of the desired range of data.
- **n_rows** (*int*) – The number of rows the data spans in the spreadsheet.
- **n_cols** (*int*) – The number of columns the data spans in the spreadsheet.

Returns: The requested scalar, 1D array, or 2D array of data.

Return type: np.float64 | npt.NDArray[np.float64]

See also

`readrngxlsx`

Call to get the DataFrame for this function.

`getrngi`

Same as this function, but returns ints.

epm_fortran.getrngi(*df: DataFrame, name: str, n_rows: Literal[1], n_cols: Literal[1]*)→ int64 [source]

epm_fortran.getrngi(*df: DataFrame, name: str, n_rows: int, n_cols: int*)→ ndarray[Any, dtype[int64]]

Provide a substitute for *subroutine getrngi* defined in nemswk1.f.

The Fortran subroutine retrieves arrays of integers from an Excel worksheet after *readrngxlsx* has already been called to read the file.

This Python version just searches through the DataFrame returned by *readrngxlsx* to find and process the requested range of data. It is returned as an array of ints with the appropriate number of dimensions.

Parameters:
- **df** (*pd.DataFrame*) – An Excel worksheet DataFrame returned by *readrngxlsx*.
- **name** (*str*) – The name of the desired range of data.
- **n_rows** (*int*) – The number of rows the data spans in the spreadsheet.

- **n_cols** (*int*) – The number of columns the data spans in the spreadsheet.

Returns: The requested scalar, 1D array, or 2D array of data.

Return type: np.int64 | npt.NDArray[np.int64]

See also

`readrngxlsx`

Call to get the DataFrame for this function.

`getrngr`

Same as this function, but returns floats.

`epm_fortran.find_xlsx_position`(*df: DataFrame, name: str, n_rows: int, n_cols: int*)→ tuple[int, int]
[source]

Find the start of a data range in a DataFrame returned by *readrngxlsx*.

The row and column indices returned point to the top left cell of the requested data range.

Parameters:

- **df** (*pd.DataFrame*) – An Excel worksheet DataFrame returned by *readrngxlsx*.
- **name** (*str*) – The name of the data range to search for.
- **n_rows** (*int*) – The number of rows the data spans in the spreadsheet.
- **n_cols** (*int*) – The number of columns the data spans in the spreadsheet.

Returns: The row index *i_row* and column index *i_col* in the DataFrame where the specified data range begins.

Return type: tuple[int, int]

Raises: **ValueError** – If the specified data range name is not found or matches more than one position in the DataFrame.

See also

`readrngxlsx`

Call to get the DataFrame for this function.

`getrngr`, `getrngi`

epm_other_ghg module

Code to account for “other” greenhouse gases and offsets.

Account for CO2 from cement and lime, read the offsets data from the GHGOFFX input file, and determine the amount of other greenhouse gas abatement using the marginal abatement curves (MACs).

`epm_other_ghg.oghg`*(restart: Restart, scedes: Scedes, variables: Variables)*→ None [source]

Compute other greenhouse gas emissions as a function of carbon price.

Calculations include lime production and clinker process CO2 emissions from the industrial module. We also handle accounting for offsets, including incentives, domestic and international offsets, and bio sequestration (if allowed). The subroutine sums abatement from covered sources at the current price, relying on the greenhouse gas marginal abatement curves (GHG MACs) as necessary.

Parameters:

- **restart** (*Restart*) – The currently loaded restart file data.
- **scedes** (*Scedes*) – The current scedes file dict.
- **variables** (*Variables*) – The active intermediate variables for EPM.

See also

`sum_emissions`

Calls this function after summing primary CO2 emissions.

`ghg_macs`

Used by this function to compute GHG MACs.

`get_file_path`

Used by this function to find the GHGOFFX file path.

`readoffsets`

Used by this function to load offsets data.

`epm_other_ghg.readoffsets`*(restart: Restart, file_path: Path)*→ None [source]

Read the greenhouse gas offsets data from the input spreadsheet.

Called by *oghg* on the first iteration of the first year to read the greenhouse gas offsets data from the "offsets" sheet of the GHGOFFX input file. The data gets stored in the appropriate arrays that are defined in the ghg common block within the ghgrep include file. The input file is an Excel spreadsheet, so this function reads it using the three functions *readrngxlsx*, *getrngr*, and *getrngi* from *epm_fortran*. They mirror the functions of the same names that are availabe in Fortran NEMS code.

Parameters:
- **restart** (*Restart*) – The currently loaded restart file data.
- **file_path** (*Path*) – Path to the XLSX file that this function should open.

See also

`oghg`

Calls this function on the first iteration of the first year.

`epm_other_ghg.ghg_macs`*(restart: Restart, variables: Variables, icat: int, p: float, year: int)*→ float
[source]

Return the amount of other greenhouse gas abatement by using the MACs.

Use the other greenhouse gas marginal abatement curves (GHG MACs) to determine the amount of abatement in a given category at a given price in a given year. This subroutine assumes that prices are organized in ascending order, and it interpolates between adjacent steps and adjacent intervals.

Parameters:
- **restart** (*Restart*) – The currently loaded restart file data.
- **variables** (*Variables*) – The active intermediate variables for EPM.
- **icat** (*int*) – Index of other greenhouse gas category.
- **p** (*float*) – Price level for abatement.
- **year** (*int*) – Four-digit calendar year.

Returns: **q** – The abatement quantity.

Return type: float

See also

`oghg`

Calls this function for MAC calculations.

`iyear5`

Used by this function to get interval numbers.

`epm_other_ghg.iyear5`*(restart: Restart, year: int)*→ tuple[int, int, float] [source]

Return the interval numbers for the MACs based on the calendar year.

This lookup utility is called by subroutine ghg_macs to retrieve the five- year interval numbers for the greenhouse gas marginal abatement curves (MACs). The interval numbers are based on the year, with lower and higher indices for each interval, as well as an interpolation fraction between them.

For example, consider the interval from 2010 to 2015; the lower index is 1, the higher index is 2, and the interpolation fraction (which is weighted on the lower side) ranges from 1.0 in 2010 to 0.2 in 2014, stepping by 0.2 each year.

Parameters:

- **restart** (*Restart*) – The currently loaded restart file data.
- **year** (*int*) – Calendar year for which to look up the interval numbers.

Returns:

- **iyl** (*int*) – Receives the lower interval number.
- **iyh** (*int*) – Receives the higher interval number.
- **fraciyl** (*float*) – Receives the interpolation fraction (weighted on the lower side) for interpolation within the five-year intervals.

See also

`ghg_macs`

Uses this function to get interval numbers.

epm_read module

Routines for reading the majority of the EPM input files.

Code in this file sets the EPM output file, reads the policy switches in the EPM control file, and parses emissions-related data from a variety of other input files. This data includes carbon taxes or caps, coal region mappings, carbon factors, mercury inputs, and California AB32 information.

`epm_read.epm_read`*(restart: Restart, scedes: Scedes, variables: Variables)*→ None [source]

Read the EPM input files and designate the EPM output file.

This function is called from NEMS main to read the contents of the EPM control file and several other EPM input files, as well as to set the EPMOUT file path for Emissions Policy Module output.

Parameters:

- **restart** (*Restart*) – The currently loaded restart file data.
- **scedes** (*Scedes*) – The current scedes file dict.
- **variables** (*Variables*) – The active intermediate variables for EPM.

See also

`find_control_file`

Used to find the EPM control file.

`read_control_file`

Used to read the contents of the EPM control file.

`get_file_path`

Used to find all other EPM input and output files.

`read_tax_or_cap`

Used read the carbon tax/cap data file.

`read_coal_regions`

Used to read the coal regions mapping file.

`read_carbon_factors`

Used to read the carbon factors file.

`read_mercury_files`

Used to read the mercury input files.

`read_ab32_files`

Used to read the California AB32 files.

`epm_read.find_control_file`(*scedes: Scedes*, *cycle: int*)→ Path [source]

Find the path to the the EPM control file using the EPMCNTLN scedes key.

The scedes value should either refer to a file inside the EPM model's input directory or be an absolute path to a file located elsewhere.

Parameters:
- **scedes** (*Scedes*) – The current scedes file dict.
- **cycle** (*int*) – The current NEMS cycle number, also known as *curirun*.

Returns: The full path to the EPM control file that the module should read.

Return type: Path

Raises: **ValueError** – If the scedes path refers to an invalid control file location as described above.

See also

`epm_read`

Calls this function.

`get_file_path`

Related function to find other EPM input and output files.

`epm_read.get_file_path`(*variables: Variables*, *file_key: str*, *output_file: bool = False*)→ Path [source]

Find the path to an input or output file other than the control file.

This function uses the input_files and output_files lists from the EPM control data stored in the intermediate variables object. Be sure to load the EPM control file before calling this function.

Parameters:
- **variables** (*Variables*) – The active intermediate variables for EPM.
- **file_key** (*str*) – The key for the specific input or output file whose path should be returned. This is the TOML key used inside the EPM control file.

- **output_file** (*bool*) – Set to True if the file path being sought is for an output file. The default is False, which indicates an input file.

Returns: The full path to the specified input or output file.

Return type: Path

Raises: **LookupError** – If the function cannot find the necessary file list from the EPM control file.

See also

`epm_read`

Calls this function.

`find_control_file`

Similar function to find the EPM control file.

`epm_read.read_control_file`*(restart: Restart, scedes: Scedes, variables: Variables, file_path: Path)*→ None [source]

Read the EPM control file.

This function is called as part of *epm_read* to handle the reading of the EPM control input file. The file contains the module debug switch, names of all other input and output files, and flags and parameters that directly control the NEMS emissions policy functionality.

In addition to loading restart file variables, this function also saves the complete contents of the control file to an attribute on the intermediate variables object.

Parameters:
- **restart** (*Restart*) – The currently loaded restart file data.
- **scedes** (*Scedes*) – The current scedes file dict.
- **variables** (*Variables*) – The active intermediate variables for EPM.
- **file_path** (*Path*) – Path to the TOML file that this function should open.

See also

`epm_read`

Calls this function.

`epm_read.read_tax_or_cap`*(restart: Restart, file_path: Path)*→ None [source]

Read the carbon tax or carbon cap input file.

Parameters:
- **restart** (*Restart*) – The currently loaded restart file data.

- **file_path** (*Path*) – Path to the CSV file that this function should open.

See also

`epm_read`

Calls this function.

`epm_read.read_coal_regions`*(restart: Restart, file_path: Path)*→ None [source]

Read the input file mapping coal regions into census divisions.

Parameters:
- **restart** (*Restart*) – The currently loaded restart file data.
- **file_path** (*Path*) – Path to the TOML file that this function should open.

See also

`epm_read`

Calls this function.

`epm_read.read_carbon_factors`*(restart: Restart, file_path: Path)*→ None [source]

Read the carbon factors input file.

Parameters:
- **restart** (*Restart*) – The currently loaded restart file data.
- **file_path** (*Path*) – Path to the TSV (tab-separated values) file that this function should open.

See also

`epm_read`

Calls this function.

`epm_read.read_mercury_files`*(restart: Restart, scedes: Scedes, parameters_file_path: Path, classes_file_path: Path, caps_file_path: Path)*→ None [source]

Read the mercury input files that EPM maintains for EMM.

Several NEMS "restart variables" that are filled as part of this function are not actually part of the restart file, so those are debug printed to the EPMOUT file to make them easier to track.

Parameters:
- **restart** (*Restart*) – The currently loaded restart file data.
- **scedes** (*Scedes*) – The current scedes file dict.

- **parameters_file_path** (*Path*) – Path to the TOML file that this function should open to find the mercury parameters.
- **classes_file_path** (*Path*) – Path to the CSV file that this function should open to find the mercury compliance classes.
- **caps_file_path** (*Path*) – Path to the CSV file that this function should open to find the mercury caps.

See also

`epm_read`

Calls this function.

`epm_read.read_ab32_files`(*restart: Restart, scedes: Scedes, parameters_file_path: Path, data_file_path: Path*)→ None [source]

Read the input files pertaining to California AB32.

Parameters:

- **restart** (*Restart*) – The currently loaded restart file data.
- **scedes** (*Scedes*) – The current scedes file dict.
- **parameters_file_path** (*Path*) – Path to the TOML file that this function should open to find the AB32 parameters.
- **data_file_path** (*Path*) – Path to the CSV file that this function should open to find the AB32 data.

See also

`epm_read`

Calls this function.

epm_regula_falsi module

Regula falsi root-finding used to solve for carbon prices in policy cases.

This file implements the *regfalsi* and regfalsibank` functions, which are used to solve for carbon prices, carbon taxes, and carbon permit prices as needed for emissions policy cases. These routines iterate in order to find the lowest price that achieves the desired emissions reduction goal.

`epm_regula_falsi.regfalsi`*(restart: Restart, variables: Variables, new_tax: float, new_sum: float)*→ float [source]

Use regula falsi root-finding to determine the new tax or permit price.

The regula falsi algorithm finds the root of a function given two points, one with positive and one with negative functional value. It has a linear convergence rate and is guaranteed to converge if the function is continuous. This function also includes some bracketing logic that heuristically attempts to rebracket the root if bracketing is lost due to anomalies in the function.

Parameters:
- **restart** (*Restart*) – The currently loaded restart file data.
- **variables** (*Variables*) – The active intermediate variables for EPM.
- **new_tax** (*float*) – Latest carbon tax rate or permit price.
- **new_sum** (*float*) – Latest sum of emitted pollutants.

Returns: New value for the carbon tax rate or permit price.

Return type: float

See also

`epm`

Calls this function to determine the carbon tax or permit price.

`regfalsibank`

Similar function which is called to set the starting price.

`epm_regula_falsi.regfalsibank`*(restart: Restart, scedes: Scedes, variables: Variables, new_tax: float, new_sum: float)*→ float [source]

Estimate the starting carbon price of a series using regula falsi.

Use a regula falsi root-finding algorithm to estimate the starting carbon price of an escalating series that results in a carbon emissions cumulative bank balance of zero. This is similar to the *regfalsi* function, but runs once per cycle prior to the first year of the program, using results (brackets) from prior cycles. The regula falsi algorithm finds the root of a function given two points, one with positive and one with negative functional value. It has a linear convergence rate and is guaranteed to converge if the function is continuous. This function includes rebracketing logic that is triggered by the REBRACKT key in the scedes, as well as some additional bracketing logic that tries to rebracket the root if bracketing is lost due to anomalies in the function.

NOTE: The epmbank common block stores the solution bracket data for this function in the restart file. If bracket data is zero, start from scratch.

Parameters:
- **restart** (*Restart*) – The currently loaded restart file data.
- **scedes** (*Scedes*) – The current scedes file dict.
- **variables** (*Variables*) – The active intermediate variables for EPM.
- **new_tax** (*float*) – Latest carbon allowance fee or tax.
- **new_sum** (*float*) – Latest carbon balance (negative if emissions are below target and positive if emissions are above target).

Returns: New value for the carbon allowance fee or tax.

Return type: float

See also

`epm`

Calls this function when needed to estimate starting carbon prices.

`regfalsibank_end`

Handles the final reporting step for this function.

`regfalsi`

Similar function which is called more frequently.

`epm_regula_falsi.regfalsibank_end`*(restart: Restart, new_tax: float)*→ float [source]

Finish reporting for the *regfalsibank* function and return *new_tax*.

This function is called in return statements within the *regfalsibank* function as a replacement for *goto 999* statements in the original Fortran. These goto statements would skip almost to the bottom of the subroutine, causing it to do its final reporting to EPMOUT and then return.

Parameters:
- **restart** (*Restart*) – The currently loaded restart file data.

- **new_tax** (*float*) – The new_tax value from regfalsibank, which is to be reported and returned.

Returns: The value of new_tax that was passed in.

Return type: float

epm_restart module

Classes for storing restart file data in memory and doing restart file I/O.

This module contains the *Restart* class, which wraps around PyFiler for reading and writing EPM-relevant parts of the NEMS restart file. In standalone runs, the restart object is used to read the restart file before EPM runs, store copies of all relevant arrays in memory while EPM is running, and then write the outputs to a new restart file once EPM finishes. In integrated runs, no actual file I/O occurs; an existing PyFiler object is directly passed in by NEMS when the restart object is created.

`epm_restart.import_pyfiler()`→ module [source]

Import and return the PyFiler module as needed for standalone runs.

This function encapsulates a number of supporting operations that ensure PyFiler actually imports and runs correctly. The first time it is called, this function assigns the imported PyFiler object to an internal global variable. Subsequent calls just return that stored object.

Returns: The imported PyFiler module object.
Return type: ModuleType

`epm_restart.get_scalar`*(array: ndarray[Any, dtype[_ScalarType_co]])*→ Any [source]

Extract the single scalar value from inside a zero-dimensional array.

Parameters: **array** (*npt.NDArray*) – An array with shape equal to ().
Returns: The single element contained inside the array.
Return type: Any
Raises: **TypeError** – If the array's shape is not equal to ().

`epm_restart.get_array`*(array: ndarray[Any, dtype[_ScalarType_co]], *, dtype: dtype[Any] | None | type[Any] | _SupportsDType[dtype[Any]] | str | tuple[Any, int] | tuple[Any, Union[SupportsIndex, collections.abc.Sequence[SupportsIndex]]] | list[Any] | _DTypeDict | tuple[Any, Any] = None)*→ ndarray[Any, dtype[_ScalarType_co]] [source]

Copy an array of values, optionally modifying the array's data type.

Parameters:
- **array** (*npt.NDArray*) – An array of values, which must be at least one-dimensional.

- **dtype** (*npt.DTypeLike, optional*) – The desired data type of the returned array. If None, then NumPy will infer the data type from the contents of the input array. This is the default behavior.

Returns: The copied array with the specified data type.

Return type: npt.NDArray

Raises: **TypeError** – If the input array is a zero-dimensional scalar array.

`epm_restart.unpack_int`(*pyfiler_variable: ndarray[Any, dtype[_ScalarType_co]]*)→ int [source]

Cautiously extract an integer value from a PyFiler variable.

Parameters: **pyfiler_variable** (*npt.NDArray*) – A PyFiler variable containing a scalar integer value.

Returns: The extracted integer value.

Return type: int

Raises: **TypeError** – If the PyFiler variable has an incompatible data type.

`epm_restart.unpack_float`(*pyfiler_variable: ndarray[Any, dtype[_ScalarType_co]]*)→ float [source]

Cautiously extract a float value from a PyFiler variable.

Parameters: **pyfiler_variable** (*npt.NDArray*) – A PyFiler variable containing a scalar float value.

Returns: The extracted float value.

Return type: float

Raises: **TypeError** – If the PyFiler variable has an incompatible data type.

`epm_restart.unpack_str`(*pyfiler_variable: ndarray[Any, dtype[_ScalarType_co]]*)→ str [source]

Cautiously extract a string value from a PyFiler variable.

Parameters: **pyfiler_variable** (*npt.NDArray*) – A PyFiler variable containing a scalar string value.

Returns: The extracted string value.

Return type: str

Raises: **TypeError** – If the PyFiler variable has an incompatible data type.

`epm_restart.unpack_int_array`(*pyfiler_variable: ndarray[Any, dtype[_ScalarType_co]]*)→ ndarray[Any, dtype[int64]] [source]

Cautiously copy an array of integers from a PyFiler variable.

Parameters: **pyfiler_variable** (*npt.NDArray*) – A PyFiler variable containing an array of integer values.

Returns: The copied array of integer values.

Return type: npt.NDArray[np.int64]

Raises: **TypeError** – If the PyFiler variable has an incompatible data type.

`epm_restart.unpack_float_array`(*pyfiler_variable: ndarray[Any, dtype[_ScalarType_co]]*)→ ndarray[Any, dtype[float64]] [source]

Cautiously copy an array of floats from a PyFiler variable.

Parameters: **pyfiler_variable** (*npt.NDArray*) – A PyFiler variable containing an array of float values.

Returns: The copied array of float values.

Return type: npt.NDArray[np.float64]

Raises: **TypeError** – If the PyFiler variable has an incompatible data type.

`epm_restart.identity`(*x: Any*)→ Any [source]

A function that returns its input.

Parameters: **x** (*Any*) – An input value.

Returns: The same input value.

Return type: Any

---

class `epm_restart.Reference`(*pyfiler_variable: ndarray[Any, dtype[_ScalarType_co]], input_converter: Callable[[Any], Any], output_converter: Callable[[Any], Any] | None = None*) [source]

Bases: `object`

A reference to a specific variable in PyFiler with data converters.

> `__init__`(*pyfiler_variable: ndarray[Any, dtype[_ScalarType_co]], input_converter: Callable[[Any], Any], output_converter: Callable[[Any], Any] | None = None*)→ None [source]

Create a new reference to a PyFiler variable with given converters.

When the restart file data comes in via PyFiler, the *input_converter* is called to transform the PyFiler variable to whatever format will be used while EPM is running. When the data is put back into the PyFiler variable after EPM runs, the *output_converter* is called to transform the data from its EPM format back to a format that can be safely assigned into the original PyFiler variable.

Parameters:

- **pyfiler_variable** (*npt.NDArray*) – The PyFiler variable that will hold the values of interest.
- **input_converter** (*Callable[[Any], Any]*) – Called on the PyFiler variable once the restart file data is available to convert the value into an EPM format.
- **output_converter** (*Callable[[Any], Any] | None, optional*) – Called on the EPM variable to convert the format back before the value is assigned into the PyFiler variable. If None (the default), then this

PyFiler variable is not used for output.

`use_for_output`()→ bool [source]

Whether this variable will be used for output back to PyFiler.

Returns: True if this variable will be used for output and False otherwise.
Return type: bool

`clone_reference`()→ Reference [source]

Make a new reference with the same PyFiler variable and converters.

Returns: A new, distinct reference object that points to the same PyFiler variable and uses the same input and output converters.
Return type: Reference

`read_values`()→ Any [source]

Read values from the PyFiler variable and apply the input converter.

Returns: Values from PyFiler after applying the input converter.
Return type: Any

`write_values`(*values: Any*)→ None [source]

Apply the output converter and write values to the PyFiler variable.

Parameters: **values** (*Any*) – New values that should be passed through the output converter and written to the PyFiler variable.
Raises: **TypeError** – If this reference is not allowed to be used for output.

class **`epm_restart.Restart`**(*pyfiler: module | None = None*) [source]

Bases: `object`

Handles restart file I/O with PyFiler and stores values while EPM runs.

This class manages the full list of restart file variables that are relevant to EPM for both input and output. It is capable of both restart file I/O with the file system (for standalone runs) and reading directly from a pre-initialized PyFiler object (for integrated runs). While EPM is running, the values for each restart file variable are stored in instance attributes whose names are given by the include file and Fortran variable name with an underscore separator, e.g., *self.ghgrep_em_resd*.

`__init__`(*pyfiler: module | None = None*)→ None [source]

Create a new restart file object and set up its instance attributes.

One instance attribute will be created for each EPM-relevant restart variable. These attributes will initially contain reference objects, which will then be dynamically replaced by values of the correct types when the restart data is read from PyFiler.

For integrated NEMS runs, a pre-initialized PyFiler object is passed in directly, but standalone runs require this class to conduct the actual PyFiler file I/O.

Parameters: **pyfiler** (*ModuleType | None, optional*) – A pre-initialized PyFiler module object to use instead of reading the restart file from disk. The default value of None indicates that PyFiler should be internally initialized and the restart file data should be read from disk.

`_find_all_references`() → list[str] [source]

List all instance attributes that are currently references.

Returns: List of strings corresponding to the names of every instance attribute that is currently an instance of the Reference class.

Return type: list[str]

`read`() → None [source]

Read the restart file data from PyFiler and set instance attributes.

If a PyFiler object was not passed in when this restart object was constructed, then the actual file read operation is carried out before any data is copied.

`write`() → None [source]

Write the instance attributes to PyFiler restart file variables.

If a PyFiler object was not passed in when this restart object was constructed, then the actual restart file write operation is carried out after all data is copied.

`_add_int`(*pyfiler_variable: ndarray[Any, dtype[_ScalarType_co]], output: bool = False*) → int [source]

Add a new Reference attribute that refers to a single integer.

Parameters:

- **pyfiler_variable** (*npt.NDArray*) – The PyFiler variable to use for reading and writing the integer value.
- **output** (*bool, optional*) – Whether this integer should be written back to PyFiler as an EPM output. Defaults to False, which indicates input only.

Returns: Reference that will be replaced with an integer when the restart file data is read from PyFiler.

Return type: int

`_add_float`(*pyfiler_variable: ndarray[Any, dtype[_ScalarType_co]], output: bool = False*)→ float [source]

Add a new reference attribute that refers to a single float.

Parameters:
- **pyfiler_variable** (*npt.NDArray*) – The PyFiler variable to use for reading and writing the float value.
- **output** (*bool*, *optional*) – Whether this float should be written back to PyFiler as an EPM output. Defaults to False, which indicates input only.

Returns: Reference that will be replaced with a float when the restart file data is read from PyFiler.

Return type: int

`_add_str`(*pyfiler_variable: ndarray[Any, dtype[_ScalarType_co]], output: bool = False*)→ str [source]

Add a new reference attribute that refers to a single string.

Parameters:
- **pyfiler_variable** (*npt.NDArray*) – The PyFiler variable to use for reading and writing the string value.
- **output** (*bool*, *optional*) – Whether this string should be written back to PyFiler as an EPM output. Defaults to False, which indicates input only.

Returns: Reference that will be replaced with a string when the restart file data is read from PyFiler.

Return type: str

`_add_int_array`(*pyfiler_variable: ndarray[Any, dtype[_ScalarType_co]], output: bool = False*)→ ndarray[Any, dtype[int64]] [source]

Add a new reference attribute that refers to an array of integers.

Parameters:
- **pyfiler_variable** (*npt.NDArray*) – The PyFiler variable to use for reading and writing the array of integer values.
- **output** (*bool*, *optional*) – Whether this array should be written back to PyFiler as an EPM output. Defaults to False, which indicates input only.

Returns: Reference that will be replaced with an array of integers when the restart file data is read from PyFiler.

Return type: npt.NDArray[np.int64]

`_add_float_array`(*pyfiler_variable: ndarray[Any, dtype[_ScalarType_co]], output: bool = False*)→ ndarray[Any, dtype[float64]] [source]

Add a new reference attribute that refers to an array of floats.

Parameters:

- **pyfiler_variable** (*Any*) – The PyFiler variable to use for reading and writing the array of float values.
- **output** (*bool, optional*) – Whether this array should be written back to PyFiler as an EPM output. Defaults to False, which indicates input only.

Returns: Reference that will be replaced with an array of floats when the restart file data is read from PyFiler.

Return type: npt.NDArray[np.float64]

`_include_parametr`() → None [source]

Create instance variables for the parametr include file.

This include file defines important global NEMS constants, or parameters, as Fortran calls them.

`_include_ncntrl`() → None [source]

Create instance variables for the ncntrl include file.

This include file contains important global NEMS control variables, which are generally set by the Integrating Module.

`_include_ab32`() → None [source]

Create instance variables for the ab32 include file.

This include file contains cap and trade variables for California AB32.

`_include_ampblk`() → None [source]

Create instance variables for the ampblk include file.

This include file contains adjustments for prices in mpblk.

`_include_bifurc`() → None [source]

Create instance variables for the bifurc include file.

This include file stores fossil energy use for entities covered (and not covered) by carbon allowance requirements.

`_include_calshr`() → None [source]

Create instance variables for the calshr include file.

This include file contains California shares of Pacific energy use for computing AB32 emissions.

`_include_ccatsdat()`→ None [source]

Create instance variables for the ccatsdat include file.

This include file contains CO2 data for the Carbon Capture, Allocation, Transportation, and Sequestration (CCATS) module.

`_include_coalemm()`→ None [source]

Create instance variables for the coalemm include file.

This include file contains data that transfers between EMM and CMM, meaning that it pertains to coal power plants.

`_include_cogen()`→ None [source]

Create instance variables for the cogen include file.

This include file contains variables related to cogeneration.

`_include_convfact()`→ None [source]

Create instance variables for the convfact include file.

This include file contains conversion factors.

`_include_ecpcntl()`→ None [source]

Create instance variables for the ecpcntl include file.

This include file contains initial input data for the ECP linear programming optimization model in EMM.

Note that the ecpcntl include file is not part of the restart file, but these variable are still accessible via PyFiler. These are here because EPM fills these variables for EMM during integrated runs.

`_include_emablk()`→ None [source]

Create instance variables for the emablk include file.

This include file contains price adjustments, which are used to add carbon fees for policies in EPM.

`_include_emeblk()`→ None [source]

Create instance variables for the emeblk include file.

This include file contains carbon emissions factors.

Note that the hgeblk common block at the bottom of this include file is not part of the restart file, but its variables are still accessible via PyFiler. They are here because EPM fills these variables for EMM during integrated runs.

`_include_emission`()→ None [source]

Create instance variables for the emission include file.

This include file contains a variety of emissions-related variables.

`_include_emoblk`()→ None [source]

Create instance variables for the emoblk include file.

This include file contains switches and policy variables for EPM.

`_include_epmbank`()→ None [source]

Create instance variables for the epmbank include file.

This include file contains variables for carbon permit pricing.

`_include_ghgrep`()→ None [source]

Create instance variables for the ghgrep include file.

This include file contains the primary emissions arrays and variables for other greenhouse gases.

`_include_hmmblk`()→ None [source]

Create instance variables for the hmmblk include file.

This include file contains variables pertaining to the Hydrogen Market Module (HMM).

`_include_indepm`()→ None [source]

Create instance variables for the indepm include file.

This include file contains CO2 process emissions that are passed from IDM to EPM.

`_include_indout`()→ None [source]

Create instance variables for the indout include file.

This include file contains outputs from IDM.

`_include_indrep`()→ None [source]

Create instance variables for the indrep include file.

This include file contains reporting variables for IDM.

`_include_lfmmout`()→ None [source]

Create instance variables for the lfmmout include file.

This include file contains refinery outputs from LFMM.

`_include_macout`()→ None [source]

Create instance variables for the macout include file.

This include file contains outputs from MAM.

`_include_mpblk`()→ None [source]

Create instance variables for the mpblk include file.

This include file contains energy prices.

`_include_ngtdmrep`()→ None [source]

Create instance variables for the ngtdmrep include file.

This include file contains NG T & D (NGTDM) report writer variables.

`_include_ogsmout`()→ None [source]

Create instance variables for the ogsmout include file.

This include file contains output variables from the oil & gas supply module, now HSM.

`_include_pmmftab`()→ None [source]

Create instance variables for the pmmftab include file.

This include file contains Ftab variables from the refinery module.

`_include_pmmout`()→ None [source]

Create instance variables for the pmmout include file.

This include file contains output variables from the refinery module.

`_include_pmmrpt`()→ None [source]

Create instance variables for the pmmrpt include file.

This include file contains output variables from the refinery module.

`_include_qblk`() → None [source]

Create instance variables for the qblk include file.

This include file contains total energy consumption quantities.

`_include_qsblk`() → None [source]

Create instance variables for the qsblk include file.

This include file contains energy consumption quantities from SEDS.

`_include_tranrep`() → None [source]

Create instance variables for the tranrep include file.

This include file contains reporting variables for TDM.

`_include_uefdout`() → None [source]

Create instance variables for the uefdout include file.

This include file contains outputs from EFD in EMM.

`_include_wrenew`() → None [source]

Create instance variables for the wrenew include file.

This include file contains renewables variables.

epm_revenue module

Functions for calculating the revenue effects of emissions policies.

This file implements the *initrev*, *accntrev*, and *epm_addoff* functions, which compute the revenue effects from any active emissions policies. This includes the revenue produced by initial allocation of permits. In a related process, we also add up the qualifying emissions offsets where needed.

`epm_revenue.accntrev`(*restart: Restart*, *scedes: Scedes*, *variables: Variables*)→ None [source]

Calculate the revenue effects of the selected emissions policy.

For a tax, this amounts to the revenue returned to the government. For an auction, it amounts to the same thing. For a market, we need to subtract out the value of the initial allocation of permits, which is done by the *initrev* function. We sum the revenue by sector:

0. Residential
1. Commercial
2. Industrial
3. Transportation
4. Utility

The revenue from each sector is summed by fuel and is directly proportional to total emissions. Emissions associated with ethanol and biodiesel are assumed to be exempt, but emissions from geothermal and MSW are included. This function also handles the case of carbon offsets (with or without bio sequestration) and the case of an allowance price maximum (safety valve).

Parameters:
- **restart** (*Restart*) – The currently loaded restart file data.
- **scedes** (*Scedes*) – The current scedes file dict.
- **variables** (*Variables*) – The active intermediate variables for EPM.

See also

`epm`

Core EPM routine that directly calls this function.

`epm_addoff`

Called by this function to account for carbon offsets.

`initrev`

Related function for computing the value of permit allocation.

`epm_revenue.initrev`*(restart: Restart)*→ None [source]

Calculate revenue produced by holding the initial allocation of permits.

The total revenue is the number of permits multiplied by the value of each permit. We sum this revenue by sector:

0. Residential
1. Commercial
2. Industrial
3. Transportation
4. Utility

Note that calling this subroutine only has an effect when the bank flag is set in EPMCNTL and the GHG banking and compliance period has begun – it otherwise updates none of the common block variables.

Parameters: **restart** (*Restart*) – The currently loaded restart file data.

See also

`epm`

Core EPM routine that directly calls this function.

`accntrev`

Related function for computing all other revenue effects.

`epm_revenue.epm_addoff`*(offset: ndarray[Any, dtype[float64]], baseyr: int, iy: int, bioseqok: int, allow_per_offset: float, offyear: int)*→ float [source]

Add up qualifying emissions offsets.

Called by *epm* to sum up qualifying emissions offsets for one year in some policy scenarios. The result depends on whether bio sequestration counts towards the goal and on the exchange rate of allowance credits per international offset (after a specified calendar year).

Parameters:
- **offset** (*npt.NDArray[np.float64]*) – The emissions offsets array–usually the array named *offset* that is defined in the *epmbank* includes file.
- **baseyr** (*int*) – Base NEMS calendar year corresponding to *curiyr* = 1.

- **iy** (*int*) – Zero-based year index for adding up offsets.
- **bioseqok** (*int*) – Set to 1 if bio sequestration offsets count towards the goal or to 0 if incentives are given but do not count towards the goal (incentive only).
- **allow_per_offset** (*float*) – Exchange rate in allowance credits per international offset.
- **offyear** (*int*) – Calendar year after which *allow_per_offset* starts being applied.

Returns: Sum of qualifying emissions offsets.

Return type: float

See also

`epm`

Calls this function in several places to add up emissions offsets.

`accntrev`

Calls this function to add up emissions offsets.

class `epm_revenue.EtaxLogic`(*restart: Restart, j: int, *, use_1987_dollars: bool = False*) [source]

Bases: `object`

Handle standard etax logic and store the etax values for each sector.

`__init__`(*restart: Restart, j: int, *, use_1987_dollars: bool = False*) → None [source]

Create an etax object with pre-computed etax values for each sector.

Parameters:
- **restart** (*Restart*) – The currently loaded restart file data.
- **j** (*int*) – Zero-based index of the current NEMS year.
- **use_1987_dollars** (*bool, optional*) – Whether the computed etax values should be given in 1987$ or not. The default is False, which yields nominal dollars.

`etax`: *float*

`tran`: *float*

`elec`: *float*

`endu`: *float*

`resd`: *float*

`comm`: *float*

epm_scedes module

Interface for handling the scedes data and accessing its key-value pairs.

The class defined in this module is a thin wrapper around a dict. It mediates EPM's access to the scedes data and also handles loading the scedes data (which may include manually parsing the scedes file from disk for standalone runs).

class **epm_scedes.Scedes**(*initializer: dict | None = None*) [source]

Bases: `object`

A wrapper class for a dict containing the scedes file information.

`__init__`(*initializer: dict | None = None*)→ None [source]

Create a new, empty scedes dict object.

After creating an empty object, call the *read* method to fill it with data. The *initializer* argument controls the data source that will be used when *read* is called.

Parameters: **initializer** (*dict* | *None*, *optional*) – The scedes data source to be used when *read* is called. If a dict is passed, then its key-value pairs will be copied. Otherwise, the default behavior is to attempt to read the scedes file from disk.

static **`_find_path`**()→ Path [source]

Try to locate the scedes file on disk for manual parsing.

Returns:	Possible path to the scedes file.
Return type:	Path

`read`()→ None [source]

Read the scedes data to finish initializing a new object.

The data source is determined by the *initializer* argument passed to the _*_init_*_ method when the object was first created.

`_load_dict`(*initializer_dict: dict*)→ None [source]

Copy the scedes data from an existing dict object in memory.

Parameters: **initializer_dict** (*dict*) – Dict to copy keys and values from. This will usually be the SCEDES attribute from the NEMS user object.

`_load_file`(*initializer_path: Path*)→ None [source]

Manually parse the scedes data from a scedes file on disk.

Parameters: **initializer_path** (*Path*) – File path to open for reading.

`get`(*key: str*, *default: str*)→ str [source]

Get the value for a scedes key, or return default if not present.

Parameters:
- **key** (*str*) – Case-insensitive scedes key.
- **default** (*str*) – Value to return if the key is not present.

Returns: Associated value from the scedes file or *default*.

Return type: str

epm_sum_emissions module

Routines for summing and reporting all United States emissions.

The contents of this file are the heart of EPM, implementing what was once the single Fortran subroutine named *sum_emissions*. Here, we add up emissions across all sectors of the economy, share emissions by region and electrical power usage, handle historical benchmarking/overwrites, and report the totals. We also read in the historical emissions from the EPMDATA input file.

`epm_sum_emissions.sum_emissions`*(restart: Restart, scedes: Scedes, variables: Variables)*→ None [source]

Sum pollutants by fuel to give aggregate totals across the country.

Emissions are determined by summing quantities of fuel use from qblk weighted by fuel-specific emissions factors from emeblk. For the current NEMS year, emissions are computed for each fuel, region, and sector and then stored in emission and ghgrep. The relevant list of fuels varies by sector, and biofuel emissions from the transportation sector are subtracted. Emissions from the electric power sector are shared out to each end-use sector based on sectors' consumption of purchased electricity. California is included as an extra region so that its emissions can be tracked for the NEMS implementation of AB32, but the California region requires some special treatment. Note: for total U.S. emissions, we add up ONLY the first 9 regions to avoid double counting (index 9 = California and 10 = United States).

The emissions are summed up following these steps in order:

1. Read historical emissions data from input file
2. Compute residential sector emissions
3. Compute commercial sector emissions
4. Compute industrial sector emissions
5. Compute transportation sector emissions
6. Compute electricity sector emissions
7. Apportion regional emissions by fuel and sector
8. Adjust covered CO2 emissions from California biofuels
9. Benchmark national emissions to history in historical years
10. Share electricity emissions to sectors and adjust totals
11. Overwrite computed emissions with history in historical years
12. Compute total emissions for use with cap-and-trade policies

13. Compute any other GHG emissions by calling subroutine oghg
14. Do reporting loop output with calls to subrotuine demand_co2

Parameters:

- **restart** (*Restart*) – The currently loaded restart file data.
- **scedes** (*Scedes*) – The current scedes file dict.
- **variables** (*Variables*) – The active intermediate variables for EPM.

See also

`epm`

Core EPM routine that directly calls this function.

`get_file_path`

Used to find the path to the emissions history file.

`read_history`

Handles reading in the historical emissions data.

`sum_emissions_residential`

Handles the residential sector emissions.

`sum_emissions_commercial`

Handles the commercial sector emissions.

`sum_emissions_industrial`

Handles the industrial sector emissions.

`sum_emissions_transportation`

Handles the transportation sector emissions.

`sum_emissions_electricity`

Handles the electric power sector emissions.

`sum_emissions_regional`

Handles regional apportionment of emissions.

`sum_emissions_california_biofuels`

Adjusts California biofuel emissions.

`sum_emissions_history_benchmark`

Handles the historical benchmarking.

`sum_emissions_share_electricity`

Shares electricity emissions to regions.

`sum_emissions_history_overwrite`

Handles the historical overwrites.

`sum_emissions_policy_totals`

Compute totals for cap-and-trade policies.

`oghg`

Called by this function to handle other greenhouse gases.

`sum_emissions_reporting_loop`

Handles the reporting loop output.

`epm_sum_emissions.read_history`(*variables: Variables, file_path: Path*)→ None [source]

Read in the historical emissions data for use by *sum_emissions*.

The historical emissions data is stored in its own EPM input file. Not sure why this historical read happens as part of *sum_emissions* and not as part of *epm_read*… possibly because the history is stored in a local EPM variable instead of a NEMS include file common block. In the future, this should likely be moved to *epm_read*, possibly gated behind a *RUNEPM != 0* conditional statement.

Parameters:
- **variables** (*Variables*) – The active intermediate variables for EPM.
- **file_path** (*Path*) – Path to the CSV file that this function should open.

See also

`sum_emissions`

Calls this function to read historical emissions data.

`epm_sum_emissions.sum_emissions_residential`(*restart: Restart, j: int, nm_resd: list[str]*)→ ndarray[Any, dtype[float64]] [source]

Compute the initial residential sector emissions for *sum_emissions*.

Parameters:
- **restart** (*Restart*) – The currently loaded restart file data.
- **j** (*int*) – Compute emissions for this zero-based year index.
- **nm_resd** (*list[str]*) – List to store the names of the residential sector fuel indicies.

Returns: **qelrs_ev** – Residential electricity consumption used to charge electric vehicles in year index *j* by region. This consumption is excluded from residential emissions and should be moved over to the transportation sector.

Return type: npt.NDArray[np.float64]

See also

`sum_emissions`

Calls this function for residential sector emissions.

`sum_emissions_transportation`

Requires the returned *qelrs_ev* value.

`sum_emissions_share_electricity`

Requires the returned *qelrs_ev* value.

`epm_sum_emissions.sum_emissions_commercial`*(restart: Restart, j: int, nm_comm: list[str])*→ ndarray[Any, dtype[float64]] [source]

Compute the initial commercial sector emissions for *sum_emissions*.

Parameters:
- **restart** (*Restart*) – The currently loaded restart file data.
- **j** (*int*) – Compute emissions for this zero-based year index.
- **nm_comm** (*list[str]*) – List to store the names of the commercial sector fuel indicies.

Returns: **qelcm_ev** – Commercial electricity consumption used to charge electric vehicles in year index *j* by region. This consumption is excluded from commercial emissions and should be moved over to the transportation sector.

Return type: npt.NDArray[np.float64]

See also

`sum_emissions`

Calls this function for commercial sector emissions.

`sum_emissions_transportation`

Requires the returned *qelcm_ev* value.

`sum_emissions_share_electricity`

Requires the returned *qelcm_ev* value.

`epm_sum_emissions.sum_emissions_industrial`*(restart: Restart, j: int, nm_indy: list[str])*→ None [source]

Compute the initial industrial sector emissions for *sum_emissions*.

Parameters:
- **restart** (*Restart*) – The currently loaded restart file data.
- **j** (*int*) – Compute emissions for this zero-based year index.
- **nm_indy** (*list[str]*) – List to store the names of the industrial sector fuel indicies.

See also

`sum_emissions`

Calls this function for industrial sector emissions.

`epm_sum_emissions.partition_cement_kiln_ccs`*(restart: Restart)*→ dict[str, float] [source]

Split up captured cement kiln carbon from combustion and non-combustion.

Captured CO2 from cement kilns in IDM is a mixture of process emissions from the calcium carbonate together with combustion emissions from all the different fossil fuels that are used to heat kilns.

Parameters: **restart** (*Restart*) – The currently loaded restart file data.

Returns: A mapping from fuel names (plus the special fuel “cement process”) to corresponding national cement kiln carbon capture totals for this year. The values will sum to the total captured cement kiln carbon that IDM sent to CCATS.

Return type: dict[str, float]

`epm_sum_emissions.sum_emissions_transportation`*(restart: Restart, j: int, nm_tran: list[str], qelrs_ev: ndarray[Any, dtype[float64]], qelcm_ev: ndarray[Any, dtype[float64]])*→ tuple[float, float] [source]

Compute the initial transportation sector emissions for *sum_emissions*.

Return *e85_ethanol_share* and *e85_gasoline_share* because they are needed for the California biofuels calculations later on in *sum_emissions*.

Parameters:
- **restart** (*Restart*) – The currently loaded restart file data.
- **j** (*int*) – Compute emissions for this zero-based year index.
- **nm_tran** (*list[str]*) – List to store the names of the transportation sector fuel indicies.

- **qelrs_ev** (*npt.NDArray[np.float64]*) – Electricity consumption by region transferred from the residential sector because it was used to charge electric vehicles.
- **qelcm_ev** (*npt.NDArray[np.float64]*) – Electricity consumption by region transferred from the commercial sector because it was used to charge electric vehicles.

Returns:

- **e85_ethanol_share** (*float*) – Fraction of all E85 consumed in year index *j* that was ethanol.
- **e85_gasoline_share** (*float*) – Fraction of all E85 consumed in year index *j* that was gasoline.

See also

`sum_emissions`

Calls this function for transportation sector emissions.

`sum_emissions_residential`

Provides the value for the *qelrs_ev* argument.

`sum_emissions_commercial`

Provides the value for the *qelcm_ev* argument.

`sum_emissions_california_biofuels`

Needs the returned *e85* variables.

`epm_sum_emissions.sum_emissions_electricity`(*restart: Restart, variables: Variables, j: int, jcalyr: int, nm_elec: list[str]*)→ None [source]

Compute the initial electric power sector emissions for *sum_emissions*.

Parameters:

- **restart** (*Restart*) – The currently loaded restart file data.
- **variables** (*Variables*) – The active intermediate variables for EPM.
- **j** (*int*) – Compute emissions for this zero-based year index.
- **jcalyr** (*int*) – Four-digit calendar year corresponding to year index *j*.
- **nm_elec** (*list[str]*) – List to store the names of the electric power sector fuel indicies.

See also

`sum_emissions`

Calls this function for electric power sector emissions.

`epm_sum_emissions.sum_emissions_regional`*(restart: Restart, j: int)*→ None [source]

Do the regional apportionment of emissions for *sum_emissions*.

Parameters:
- **restart** (*Restart*) – The currently loaded restart file data.
- **j** (*int*) – Apportion regional emissions for this zero-based year index.

See also

`sum_emissions`

Calls this function for regional emissions apportionment.

`epm_sum_emissions.sum_emissions_california_biofuels`*(restart: Restart, j: int, e85_ethanol_share: float, e85_gasoline_share: float)*→ None [source]

Adjust the California biofuels emissions numbers for *sum_emissions*.

Parameters:
- **restart** (*Restart*) – The currently loaded restart file data.
- **j** (*int*) – Adjust California biofuels emissions for this zero-based year index.
- **e85_ethanol_share** (*float*) – Fraction of all E85 consumed in year index *j* that was ethanol. This value is computed and returned by the *sum_emissions_transportation* function.
- **e85_gasoline_share** (*float*) – Fraction of all E85 consumed in year index *j* that was gasoline. This value is computed and returned by the *sum_emissions_transportation* function.

See also

`sum_emissions`

Uses this function to adjust California biofuels emissions.

`sum_emissions_transportation`

Returns *e85* values used by this function.

`epm_sum_emissions.sum_emissions_history_benchmark`*(restart: Restart, variables: Variables, j: int, iy: int)*→ None [source]

Benchmark the computed emissions values to history for *sum_emissions*.

Parameters:

- **restart** (*Restart*) – The currently loaded restart file data.
- **variables** (*Variables*) – The active intermediate variables for EPM.
- **j** (*int*) – Benchmark the emissions for this zero-based year index.
- **iy** (*int*) – Year index – should be set to the same value as *j*.

See also

`sum_emissions`

Uses this function for benchmarking in historical years.

`epm_sum_emissions.sum_emissions_share_electricity`*(restart: Restart, j: int, qelrs_ev: ndarray[Any, dtype[float64]], qelcm_ev: ndarray[Any, dtype[float64]])*→ None [source]

Share electric power sector emissions to sectors for *sum_emissions*.

Parameters:

- **restart** (*Restart*) – The currently loaded restart file data.
- **j** (*int*) – Share out emissions for this zero-based year index.
- **qelrs_ev** (*npt.NDArray[np.float64]*) – Electricity consumption by region transferred from the residential sector to the transportation sector because it was used to charge electric vehicles.
- **qelcm_ev** (*npt.NDArray[np.float64]*) – Electricity consumption by region transferred from the commercial sector to the transportation sector because it was used to charge electric vehicles.

See also

`sum_emissions`

Calls this function to share out electricity emissions.

`sum_emissions_residential`

Provides the value for the *qelrs_ev* argument.

`sum_emissions_commercial`

Provides the value for the *qelcm_ev* argument.

`epm_sum_emissions.sum_emissions_history_overwrite`*(restart: Restart, variables: Variables, j: int)*→ None [source]

Overwrite computed emissions with history as needed by *sum_emissions*.

Parameters:

- **restart** (*Restart*) – The currently loaded restart file data.
- **variables** (*Variables*) – The active intermediate variables for EPM.

- **j** (*int*) – Overwrite with history for this zero-based year index.

See also

`sum_emissions`

Calls this function to overwrite emissions with history.

`epm_sum_emissions.sum_emissions_policy_totals`(*restart: Restart*, *j: int*)→ None [source]

Add up the cap-and-trade policy CO2 totals for *sum_emissions*.

Parameters:
- **restart** (*Restart*) – The currently loaded restart file data.
- **j** (*int*) – Compute emissions policy totals for this zero-based year index.

See also

`sum_emissions`

Calls this function to get totals for emissions policies.

`epm_sum_emissions.sum_emissions_reporting_loop`(*restart: Restart*, *variables: Variables*, *nm_resd: list[str]*, *nm_comm: list[str]*, *nm_indy: list[str]*, *nm_tran: list[str]*, *nm_elec: list[str]*, *regnam: list[str]*)→ None [source]

Handle the NEMS reporting loop output for *sum_emissions*.

This part of the *sum_emissions* process uses repeated calls to the *demand_co2* function to write the final emissions values to a file in CSV format.

Parameters:
- **restart** (*Restart*) – The currently loaded restart file data.
- **variables** (*Variables*) – The active intermediate variables for EPM.
- **nm_resd** (*list[str]*) – The names of the residential sector fuel indicies.
- **nm_comm** (*list[str]*) – The names of the commercial sector fuel indicies.
- **nm_indy** (*list[str]*) – The names of the industrial sector fuel indicies.
- **nm_tran** (*list[str]*) – The names of the transportation sector fuel indicies.
- **nm_elec** (*list[str]*) – The names of the electric power sector fuel indicies.
- **regnam** (*list[str]*) – The names of the NEMS regions by region index.

See also

`sum_emissions`

Calls this function for output during the reporting loop.

`get_file_path`

Used to find the path to the DEMAND_CO2 output file.

`demand_co2`

Used by this function to write out CO2 emissions values.

`epm_sum_emissions.demand_co2`(*restart: Restart, iunit: TextIO, emis: ndarray[Any, dtype[float64]], fuel: ndarray[Any, dtype[float64]], name: list[str], sector: str, n_pet: int, n_coal: int, n_gas: int, n_other: int, n_elec: int, i_reg: int*)→ None [source]

Report CO2 emissions and energy consumption for one sector and region.

Called by the *sum_emissions_reporting_loop* function to output CO2 emissions and the corresponding energy use from which they were computed. One call to this function handles the reporting for all fuels across one sector in one region. The CSV-formatted report is written to the output file whose file handle is passed in.

NOTE: This function assumes that fuels are indexed in five clusters: first come all of the petroleum products, then the coal, then the natural gas, then the others, and finally the purchased electricity. Make sure that the input arrays adhere to this pattern or the reported subtotals will be incorrect.

Parameters:

- **restart** (*Restart*) – The currently loaded restart file data.
- **iunit** (*TextIO*) – File handle of the DEMAND_CO2 output file.
- **emis** (*npt.NDArray[np.float64]*) – Sector CO2 emissions array.
- **fuel** (*npt.NDArray[np.float64]*) – Sector fuel consumption array with same dimensions as *emis*.
- **name** (*list[str]*) – List of fuel index names with length matching the first dimension of *emis* and *fuel*.
- **sector** (*str*) – Name of the sector, e.g., ‘Residential’.
- **n_pet** (*int*) – Number of fuel indices that are petroleum products.
- **n_coal** (*int*) – Number of fuel indices that are coal.
- **n_gas** (*int*) – Number of fuel indices that are natural gas.
- **n_other** (*int*) – Number of fuel indices that are “other” fuels.
- **n_elec** (*int*) – Number of fuel indices that are purchased electricity. This will be 0 for the electric power sector and 1 for all other sectors.
- **i_reg** (*int*) – Report results for the region at this index.

Raises: **TypeError** – If any of the input arrays do not have the expected dimensions.

See also

`sum_emissions_reporting_loop`

Calls this function.

epm_variables module

Handling of static variables that EPM expects will be saved between calls.

The small amount of code in this module exists only to create the collection of variables and initialize them to sensible values.

class `epm_variables.Variables`(*restart: Restart*) [source]

Bases: `object`

Stores intermediate variables that need to be saved between EPM runs.

All stored EPM variables are directly accessible as instance attributes.

`__init__`(*restart*: *Restart*)→ None [source]

Create a new collection of initialized variables.

The individual variables are initialized to zero, False, or some other “null” value of the appropriate type.

Parameters: **restart** (*Restart*) – The currently loaded restart file data.

run_epm module

Primary interface for running EPM with its supporting Python framework.

The only required argument (*mode*) specifies whether to run the *epm* function only (“main”), the *epm_read* function only (“read”), both the *epm_read* and *epm* functions in sequence (“both”), or neither function (“none”). In every mode, the restart file object, scedes file object, and intermediate variables object will all be initialized to start and cleaned up afterward.

class `run_epm.Mode`(*value, names=None, *, module=None, qualname=None, type=None, start=1, boundary=None*) [source]

Bases: `StrEnum`

Constants for each EPM mode that can be specified via command line.

When running the Emissions Policy Module, this string enum communicates which code should be executed. The available options are to run only the core *epm* routine (“main”), only the *epm_read* routine (“read”), both routines in sequence (“both”), or neither routine (“none”).

The surrounding framework code for the scedes file, restart file, and EPM intermediate variables are always executed.

`MAIN`= *'main'*

`READ`= *'read'*

`BOTH`= *'both'*

`NONE`= *'none'*

`run_epm.get_args`()→ Namespace [source]

Get the command line arguments and parse them with an arg parser.

The only required command line argument is *mode*, which specifies whether we should run *epm* only, *epm_read* only, both functions, or neither. The optional arguments *cycles*, *years*, and *iters* allow for overriding the NEMS control variables *numiruns*, *lastyr* (as well as *lastcalyr*), and *maxitr* instead of using the values from the input restart file.

Returns: Namespace containing parsed command line arguments.

Return type: argparse.Namespace

---

class `run_epm.Arguments`(*mode: Mode, *, cycles: int | None = None, years: int | None = None, iters: int | None = None*) [source]

Bases: `object`

A simple structure to store and validate the external arguments.

`__init__`(*mode: Mode, *, cycles: int | None = None, years: int | None = None, iters: int | None = None*)→ None [source]

Create a new arguments object with the given argument values.

Parameters:

- **mode** (*Mode*) – String enum specifying which of the primary EPM routines to run. See the Mode type for more details.
- **cycles** (*int | None, optional*) – If not None, this overrides the restart file value of *numiruns*.
- **years** (*int | None, optional*) – If not None, this overrides the restart file value of *lastyr*. The value of *lastcalyr* is also overridden appropriately.
- **iters** (*int | None, optional*) – If not None, this overrides the restart file value of *maxitr*.

See also

`Mode`

StrEnum subtype for the *mode* argument to this function.

`validate`(*restart: Restart*)→ None [source]

Use the restart file data to validate the external arguments to EPM.

Parameters: **restart** (*Restart*) – The restart file object, which should already be loaded with data from the input restart file.

Raises:

- **ValueError** – If the *cycles* argument is less than 1.
- **ValueError** – If the *years* argument is less than 1 or greater than *mnumyr*.
- **ValueError** – If the *iters* argument is less than 1.

`run_epm.print_run_message`(*args: Arguments, restart: Restart, *, integrated: bool*)→ None [source]

Print a run message including the mode and some restart variables.

Parameters:

- **args** (*Arguments*) – Structure containing the external arguments sent to the EPM code.
- **restart** (*Restart*) – The loaded restart file data object.
- **integrated** (*bool*) – Whether EPM is running in integrated (True) or standalone (False) configuration.

`run_epm.run_epm_integrated`(*args: Arguments, restart: Restart, scedes: Scedes, variables: Variables*)→ None [source]

Run the core EPM code once in integrated NEMS run configuration.

Parameters:

- **args** (*Arguments*) – Structure containing the external arguments sent to the EPM code.
- **restart** (*Restart*) – The restart file object, which should already be loaded with data from the input restart file.
- **scedes** (*Scedes*) – The scedes dict object, which should already be loaded with key-value pairs from the keys.sed file.
- **variables** (*Variables*) – The intermediate variables for EPM, which should already be loaded from the pickle file if the file exists.

`run_epm.run_epm_standalone`(*args: Arguments, restart: Restart, scedes: Scedes, variables: Variables*)→ None [source]

Run the core EPM code through all loops in standalone configuration.

Parameters:

- **args** (*Arguments*) – Structure containing the external arguments sent to the EPM code.
- **restart** (*Restart*) – The restart file object, which should already be loaded with data from the input restart file.
- **scedes** (*Scedes*) – The scedes dict object, which should already be loaded with key-value pairs from the keys.sed file.
- **variables** (*Variables*) – The intermediate variables for EPM, which should already be loaded from the pickle file if the file exists.

`run_epm.run_epm`(*mode: Mode, pyfiler: module | None = None, user: SimpleNamespace | None = None, *, cycles: int | None = None, years: int | None = None, iters: int | None = None*)→ None [source]

Primary interface function for executing the NEMS EPM code as a whole.

When this file is executed as a command line script, it calls this function and runs EPM standalone, relying on PyFiler internally for restart file IO. For integrated NEMS runs, this function can be imported and directly passed a PyFiler module object as input. When running integrated, the *cycles*, *years*, and *iters* arguments are ignored.

Parameters:

- **mode** (*Mode*) – String enum specifying which of the primary EPM routines to run. See the Mode type for more details.

- **pyfiler** (*ModuleType* | *None*, *optional*) – A pre-initialized PyFiler module object to use for restart file data. This is used for integrated NEMS runs and should be set to the default value of None for standalone runs when EPM will need to manage PyFiler on its own.
- **user** (*SimpleNamespace* | *None*, *optional*) – The NEMS user object. For integrated runs, this will be used to access the scedes dict and to store internal EPM variables. Defaults to None, which is appropriate for standalone runs.
- **cycles** (*int* | *None*, *optional*) – If not None, this overrides the restart file value of *numiruns*. The default is to use the restart file value.
- **years** (*int* | *None*, *optional*) – If not None, this overrides the restart file value of *lastyr*. The value of *lastcalyr* is also overridden appropriately. The default is to use the restart file values.
- **iters** (*int* | *None*, *optional*) – If not None, this overrides the restart file value of *maxitr*. The default is to use the restart file value.

See also

`Mode`

StrEnum subtype for the first argument to this function.