Carbon Capture, Allocation, Transportation, and Sequestration (CCATS) 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

The U.S. Energy Information Administration (EIA), the statistical and analytical agency within the U.S. Department of Energy (DOE), prepared this report. By law, our data, analyses, and forecasts are independent of approval by any other officer or employee of the U.S. Government. The views in this report do not represent those of DOE or any other federal agencies.

CCATS Documentation

The **C**arbon **C**apture, **A**llocation, **T**ransportation and **S**equestration (**CCATS**) module represents the carbon capture industry in the National Energy Modeling System (NEMS). CCATS was developed by the U.S. Energy Information Adminstration (EIA) for the Annual Energy Outlook (AEO).

This section provides an overview of the model, and describes the assumptions, inputs, and model formulation.

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

Table of Contents

Introduction

The Carbon Capture, Allocation, Transportation, and Sequestration (CCATS) module models the captured carbon dioxide (CO_2) system within the National Energy Modeling System (NEMS). CCATS endogenously allocates and transports the projected supply of captured CO_2 from NEMS modules to utilization and storage sites throughout the United States.

At its core, CCATS is an optimization model that minimizes various operation and investment costs for capturing, transporting, and sequestering or utilizing CO_2. After applying policy incentives, the module determines the most cost-effective network flow of CO_2 from supply sources to demand locations and projects the development of CO_2 infrastructure for both transportation and saline storage until 2050.

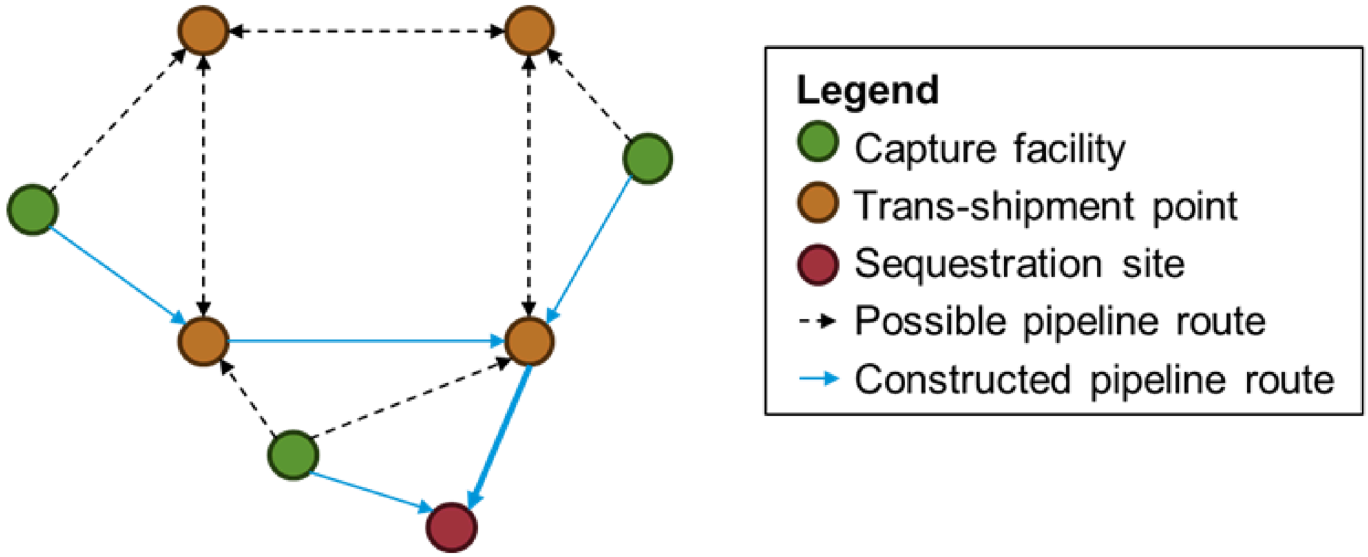
CCATS was first introduced in NEMS for the Annual Energy Outlook 2025 (AEO2025) to better reflect the emerging market for captured carbon dioxide (CO_2). Prior to the Inflation Reduction Act (IRA), policy incentives for carbon capture and storage were insufficient to support the development of carbon storage at scale. The module was designed to be flexible to incorporate future policies and to more accurately project potential long-term trends in U.S. energy markets. CCATS replaced the Capture, Transport, Utilization, Storage (CTUS) from prior AEOs.

Model Overview

CCATS represents three distinct components of CO_2 flow as interconnected nodes in a network (illustrated in Figure 1).

1. **Capture facilities**: Facilities where CO_2 is captured
2. **Trans-shipment points**: A pipeline network that connects capture sites to sequestration locations, including both existing infrastructure and potential expansion routes.
3. **Sequestration sites**: Destinations where CO_2 is stored

Figure 1. CCATS nodal network representation



Capture facility nodes represent sources of CO_2 supply from other modules in NEMS. Specifically, CCATS receives quantities of captured CO_2 from electric power generation, ethanol production, natural gas processing, hydrogen production, and cement production. CCATS currently does not represent direct air capture (DAC).

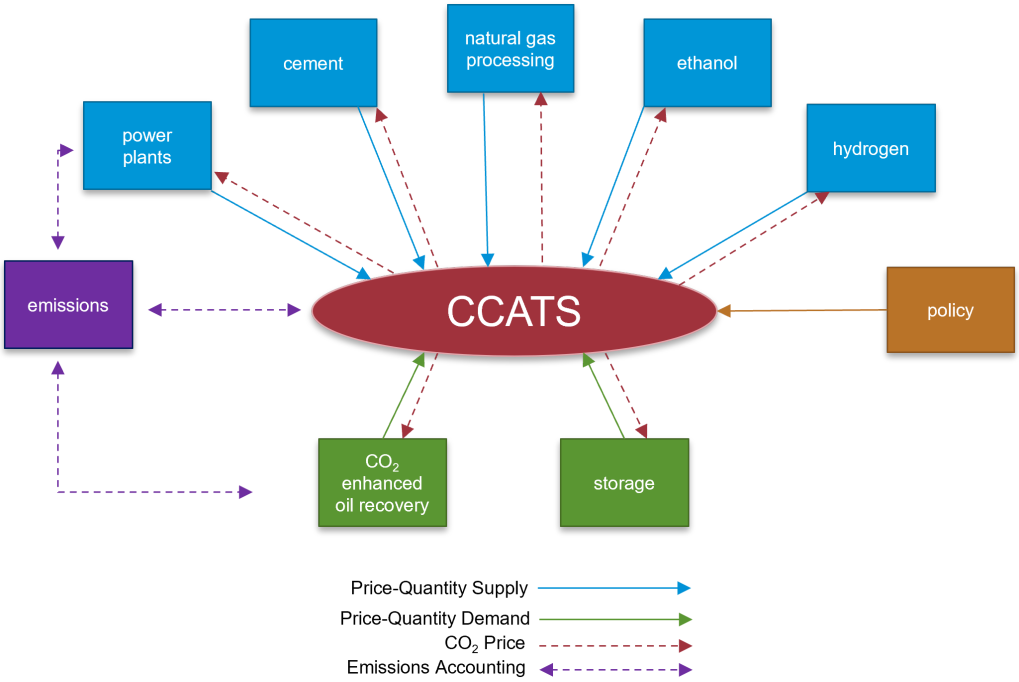
CO_2 demand in CCATS comes from either CO_2 enhanced oil recovery (EOR) wells or storage in saline formations. Today, the overwhelming majority of captured CO_2 is directed toward CO_2 EOR, a process in which CO_2 is injected into oil and natural gas wells to extract additional hydrocarbon resources. Demand from other sources of CO_2 utilization such as the food and beverage industry and electrofuels, or e-fuels, are currently not modeled in CCATS.

CCATS accounts for both operating and investment costs for capacity expansion for trans-shipment and sequestration node types. Note that capture costs are represented in other NEMS modules and have already been taken into account when CO_2 supply is received from other modules.

The model optimizes the flow of CO_2 from supply sources to sequestration sites using a linear program that minimizes total system costs while incorporating applicable tax credits and other revenues as negative costs. The model solution determines optimal transportation routes and sequestration locations. The model solution also provides CO_2 prices, which are passed off to NEMS modules and inform their carbon capture and investment decisions in equilibrium.

The interaction between CCATS and the other NEMS modules is shown in Figure 2.

Figure 2. Overview of CCATS interaction with other modules in NEMS



Geographic Representation

CCATS represents the three main geographical areas where current carbon capture and sequestration operations are active in the U.S.: the Gulf Coast, the Permian Basin, and the Rocky Mountains/Great Plains. Each of these markets is local, and no existing pipelines move CO_2 between these regions. CCATS is designed to build on this local transportation infrastructure to support additional volumes.

Model Assumptions

Capture Facilities

CCATS makes various assumptions on the granularity, geographic location, and costs associated with captured CO_2 supplies received from NEMS for the following industries and corresponding NEMS modules:

- Electric power generation - Electricity Market Module (EMM)
- Ethanol production - Liquid Fuels Market Module (LFMM)
- Natural Gas Processing - Hydrocarbon Supply Module (HSM)
- Hydrogen Production - Hydrogen Market Module (HMM)
- Cement production - Industrial Demand Module (IDM)

CCATS receives CO_2 supplies from NEMS at either at the census region or census division level. However, the CCATS optimization model operates on a more granular level, specifically at the discrete facility level, to provide more accurate projections geographically. Accordingly, CCATS disaggregates captured CO_2 supply to specific CO_2 supply facilities using the following methodology.

First, we assign CO_2 supply to facilities with existing infrastructure to represent observed CO_2 in the data. Second, as the captured CO_2 industry grows with volumes beyond the capacity of existing facilities, CCATS ranks facilities based on estimated costs to install capture technology and costs to connect supply facilities to the pipeline network. This determines which facilities will find it most economical to invest in capturing CO_2 first.

Installation cost assumptions vary based on industry and the availability of expert studies and analysis. For natural gas power plants, coal power plants, and bioenergy with carbon capture and storage power plants in the electric power sector, we use modified versions of NETL power plant studies. [1], [2] These data provide the locations, expected cost of capture and estimated CO_2 capture potential of existing power plants suitable for carbon capture retrofit.

For ethanol, natural gas processing, hydrogen (represented by ammonia), and cement facilities, we use the NETL Industrial Carbon Capture Retrofit Database [3] to identify facilities suitable for retrofit with carbon capture. We subsequently combine estimated capture cost and CO_2 capture potential from this dataset with geographic location data from EPA's Greenhouse Gas Reporting Program. [4] We also use EPA Subpart PP [5] and an analysis by CATF [6] to determine whether a facility has been capturing CO_2, and if so, for how long. Finally, we make modifications to assessed CO_2 capture potential based on EIA-64A, EIA-757, and EIA-816 survey data.

In addition to investment in existing facilities, CCATS also has the option to install carbon capture at new facilities as the industry further grows. Characteristics of these new facilities and their corresponding capture costs are provided by the other NEMS modules to CCATS as input parameters.

To save on runtime, facilities with a capture cost greater than $70/MMmT ($2023) are excluded from the optimization.

Table 1. CO_2 capture potential at represented existing facilities for the optimization

Census Division	Ammonia	Cement	Coal Power Plant	Ethanol	Natural Gas Power Plant	Natural Gas Processing
New England		0.3	8.8		36.4	
Mid Atlantic		5.1	67.1	0.5	165.8	
East North Central	0.8	8.2	303.1	12	219.5	0.5
West North Central	2.5	14.1	425.7	31.5	25.8	0
South Atlantic	1.1	13.7	444.1		238.9	
East South Central	0.9	6.4	214.5	1	147.8	
West South Central	9.7	12.7	326.9	1.4	332.4	9.7
Mountain	0.6	8	219.1		120.4	3.3
Pacific	0.1	10			121.1	
Total	15.6	78.6	2,009.30	46.4	1,408.10	13.6

Source: National Energy Technology Laboratory: Schmitt *et al.*[1], Buchheit *et al.*[2], Hughes *et al.*[3]; U.S. Environmental Protection Agency: [4], [5], Clean Air Task Force: [6].

Pipeline Network

Nodal Map

Captured CO_2 is transported from capture sites to either EOR or saline storage via pipelines. In CCATS, CO_2 can be transported directly from a supply source to a sequestration site, or indirectly via a series of trans-shipment points. This representation reflects current industry dynamics where some smaller CO_2 supply sites send captured CO_2 to a single storage or EOR site, while other groups of CO_2 capture infrastructure are connected via a regional pipeline network.

We build our transportation network by first representing the existing U.S. CO_2 pipeline network as trans-shipment points on the U.S. map. We add to this set a uniform grid of nodes representing the potential trans-shipment network that can be built for capacity expansion. Finally, we include all the CO_2 capture sites, CO_2 EOR sites, and saline formation storage sites to the network.

We connect the all the various nodes then limit the set of connections used in the model based on pipeline length and node type. For example, sequestration nodes cannot connect to other sequestration nodes.

Cost Assumptions

To calculate installation and operations costs, we first group the set of connected nodes by pipeline length and region. All connections that are of similar distances and are in the same region use the same cost assumptions.

Regionalized pipeline costs are based on the FECM/NETL CO_2 Transport Cost Model [7], modifying a natural gas pipeline study from Brown *et al.*[8] to account for the higher costs of CO_2 pipelines. This model is highly granular and includes information on operating and financing costs by pipeline diameter, length, and pump count.

Table 2. Select cost curves by region from Brown et al

Pipeline Region	**Pipeline Length (miles)**	**Cost Curve Slope ($/tonne CO_2)**
New England	150	\$51.42
Great Plains	150	\$23.47
New England	400	\$142.93
Great Plains	400	\$68.67

Source: U.S. Energy Information Administration.

For each pipeline length, we apply cost factors from the FECM/NETL study to combinations of pipeline diameters and pump counts. We assume a 20-year project lifespan. This yields various cost possibilities for transporting a certain volume of CO_2 by a certain distance. We separate these total costs into electricity costs, fixed operating and maintenance costs, and capital costs.

To obtain installation cost parameters, we choose the least costly option in terms of fixed and capital costs based on the previous calculation. Based on this cost curve, we run a linear regression to produce the installation cost linear parameters provided to the model.

Variable costs include both maintenance costs and electricity costs. We calculate electricity operating costs based on the maximum flowrate for each diameter-pump-length combination, and some assumptions on pump requirements. Specifically, we treat CO_2 that is within the pipelines as a supercritical fluid, modeling the fluid as incompressible. We assume pump stations are built along the pipeline at a frequency of no more than two pumps per 100 miles. We then calculate pump power requirements and total electricity costs using electricity prices received endogenously from NEMS.

Table 3. Select maximum flow rates (MMtonne/year)

Pipeline Length (Miles)	**Number of Pumps**	**Diameter**			
		12 in	**24 in**	**36 in**	**48 in**

Pipeline Length (Miles)	Number of Pumps	Diameter			
		12 in	**24 in**	**36 in**	**48 in**
150	0	2.5	15.15	43.39	91.45
150	1	3.54	21.48	61.48	129.53
150	2	4.35	26.33	75.36	158.75
400	0	1.52	9.23	26.47	55.81
400	1	2.16	13.11	37.54	79.13
400	2	2.65	16.08	46.04	97.02

Source: U.S. Energy Information Administration.

In addition to installation and operating costs, we add cost multipliers to any pipelines that cross over water, or over land but is covered under the National Park Service [9] or National Register of Historic Places. [10] These multipliers account for rerouting or additional permitting costs associated with these routes.

Saline storage assumptions

Saline formations are the only storage option for CO_2 in CCATS. To accurately model CO_2 storage, we calculate the amount of CO_2 that can be stored in each formation, and the costs of setting up an injection site, the process of injecting CO_2, and storing CO_2 in the formation.

To do this, we relied on the FE/NETL CO_2 Saline Storage Cost Model [11] for a comprehensive list of geologic formations, as well as the base geologic/engineering calculations for injection rates, and maximum CO_2 storage amounts in the formations. The model was also used to estimate the costs for each individual injection project. A summary of the storage formations that are input into the model are shown below. The full list of storage formations and their characteristics can also be donwloaded here: `table-storage-formations-full.csv`.

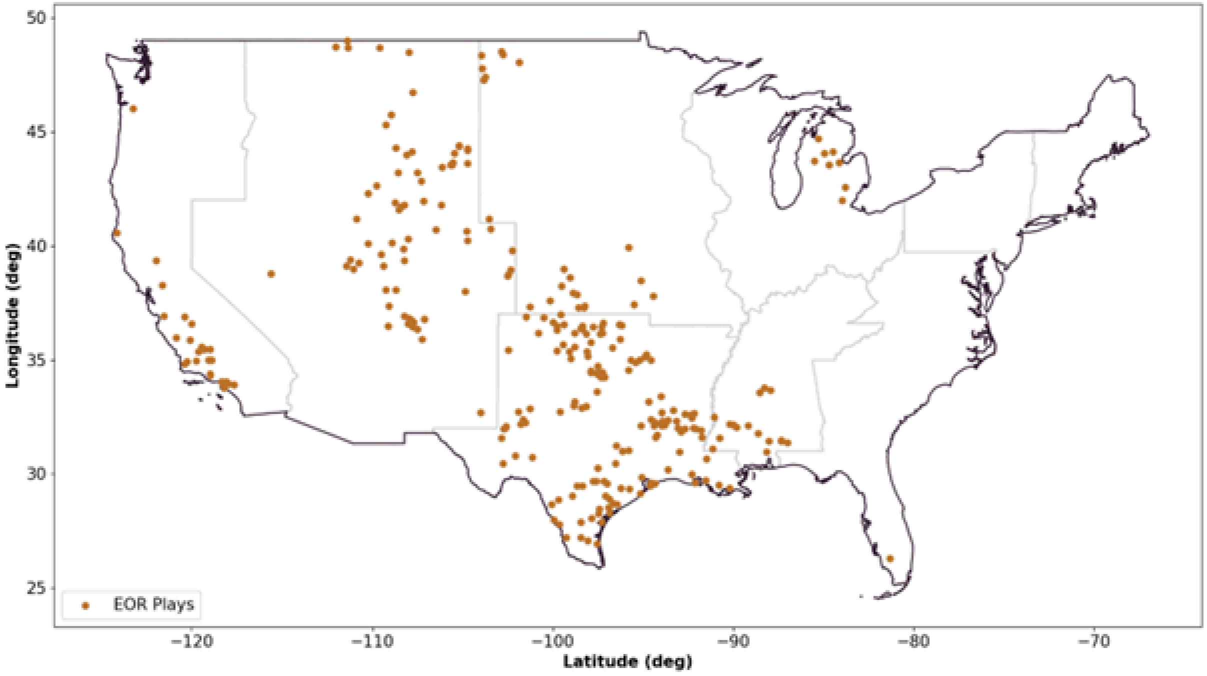
Table 4. Summary of Storage Formations

	South	Midwest	West	Mideast	Northeast
Area (sq miles)	675,703	262,009	375,353	17,128	8,201
Average maximim CO2 per injection project (Mmtonnes)	7,783,104	5,679,189	8,472,865	4,021,910	5,037,070
Maximum number of injection projects	9,145	3,172	4,357	90	30
Average injection rate per project (Mmtonnes/project/year)	2,334,931	1,135,838	1,694,573	104,064	167,902

Source: U.S. Energy Information Administration.

CO_2 EOR assumptions

Maximum demand for captured CO_2 from EOR sites is provided at the geological formation level by HSM. CCATS is not required to meet all CO_2 demanded for EOR because CCATS currently does not represent natural sources of CO_2. Note that natural sources of CO_2 fulfilled 62% of CO_2 supplied to EOR in 2023. [12]



Source: U.S. Energy Information Administration.

Price assumptions

CO_2 prices are calculated after the CCATS linear program has solved. Specifically, we calculate a regional volume-weighted average of the shadow prices produced by the model. This price is inclusive of transportation and sequestration costs, net policy revenue and revenue from selling CO_2 to EOR sites. This price does not include capture costs, as these costs are calculated by the NEMS modules that interface with CCATS as part of their carbon capture decisions.

Technology improvement rate assumptions

CCATS includes a technology improvement rate that reduces the cost of a technology over time. A report by Fahs *et al.*[13] at DOE estimates that major cost reductions are possible for carbon capture, but only moderate and small reductions for transport and storage, respectively. As such, we include an annual improvement rate of 1% for pipeline transport and saline storage.

Capacity expansion and financing assumptions

Assumptions for transportation and storage infrastructure investments are listed in Table 5. The fixed O&M fraction is the relative amount of fixed operation and maintenance costs within as compared with the capital investment cost. The buffer assumption is the amount of extra capacity that must be built.

Table 5. CCATS Financing Assumptions

Parameter	Value
Debt ratio	40%
Return over capital cost	5%
Risk premia	2%
Financing years - Transportation	20
Financing years - Storage	26
Fixed O&M - Transportation	2.5%
Fixed O&M - Storage	8.7%
Capacity Expansion Buffer	15%

Source: U.S. Energy Information Administration.

Legislation and Regulations

In representing existing policy, CCATS focuses on the expansion and enhancement of 45Q tax credits in the following three legislative acts.

Energy Improvement and Extension Act of 2008

The Energy Improvement and Extension Act of 2008 [14] included the establishment of the 45Q tax credit for the capture and sequestration of CO_2 from industrial facilities. This law established that CO_2 must be captured and disposed of within the United States.

Bipartisan Budget Act of 2018

The Bipartisan Budget Act of 2018 [15] included extending the availability of the 45Q tax credit to facilities that began construction before 2024 and increased the tax credit. For EOR, the tax credit began at $10/mT and increased to $35/mT in 2027. For saline storage, the tax credit began at $20/mT and increased to $50/mT in 2027. After 2027, the tax credit is inflation adjusted. The tax credit is available for 12 years.

Inflation Reduction Act of 2022

The Inflation Reduction Act (IRA) of 2022 [16] extended the 45Q tax credit to eligibly facilities that begin construction before 2032 and meet minimum quantity thresholds. IRA increased the tax credits from previous legislations to $60 per metric ton for captured CO_2 sent to EOR sites, and to $85 per metric ton for captured CO_2 permanently sent to geologic storage sites. The cax credits last for 12 years after the carbon capture equipment associated with the project is placed into service. Tax credits are adjusted for inflation starting in 2027 and are indexed to 2025 as the base year.

Sources

[1](1,2) Tommy Schmitt, Sarah Leptinsky, Marc Turner, Alexander Zoelle, Charles W. White, Sydney Hughes, Sally Homsy, Mark Woods, Hannah Travis Shultz, and Robert E. James III. Cost and performance baseline for fossil energy plants volume 1: bituminous coal and natural gas to electricity. Technical Report, National Energy Technology Laboratory (NETL), 2022. URL: https://www.osti.gov/servlets/purl/1893822/.

[2](1,2) Kyle L. Buchheit, Alex Zoelle, Eric Lewis, Marc Turner, Tommy Schmitt, Norma Kuehn, Sally Homsy, Shannon McNaul, Sarah Leptinsky, Allison Guinan, Mark Woods, Travis Shultz, Timothy Fout, and Gregory Hackett. Eliminating the derate of carbon capture retrofits (rev. 2). Technical Report, National Energy Technology Laboratory (NETL), 2023. URL: https://www.osti.gov/servlets/purl/1968037/.

[3](1,2) Sydney Hughes, Alex Zoelle, Mark Woods, Samuel Henry, Sally Homsy, Sandeep Pidaparti, Norma Kuehn, Hannah Hoffman, Katie Forrest, Timothy Fout, William Summers, Steve Herron, and Eric Grol. Industrial CO\textsubscript 2 capture retrofit database (IND CCRD). Technical Report, National Energy Technology Laboratory (NETL), 2023. URL: https://netl.doe.gov/energy-analysis/details?id=42be56f6-f37e-46f4-88ea-a77868da32f3.

[4](1,2) U.S. Environmental Protection Agency. Greenhouse gas reporting program (GHGRP) FLIGHT. URL: https://www.epa.gov/ghgreporting.

[5](1,2) U.S. Environmental Protection Agency. Subpart pp – suppliers of carbon dioxide. Accessed February 5, 2025. URL: https://www.epa.gov/ghgreporting/subpart-pp-suppliers-carbon-dioxide.

[6](1,2) Clean Air Task Force. Us carbon capture activity and project table. Accessed February 5, 2025. URL: https://www.catf.us/ccstableus/ (visited on 2025-02-05).

[7] David Morgan, Allison Guinan, and Alana Sheriff. FECM/NETL CO\textsubscript 2 transport cost model (2023): description and user's manual. Technical Report, National Energy Technology Laboratory (NETL), U.S. Department of Energy, 2023. doi:10.2172/1992905.

[8] Daryl Brown, Krishna Reddi, and Amgad Elgowainy. The development of natural gas and hydrogen pipeline capital cost estimating equations. *International Journal of Hydrogen Energy*, 47:33813–33826, 2022.

[9] U.S. General Services Administration. National park boundaries. Data.Gov, December 2020. Last modified December 2, 2020. Accessed February 5, 2025. URL: https://catalog.data.gov/dataset/national-park-boundaries (visited on 2025-02-05).

[10] U.S. General Services Administration. National register of historic places. Data.Gov, November 2014. Last modified November 2, 2014. Accessed February 5, 2025. URL: https://catalog.data.gov/dataset/national-register-of-historic-places-7eebd (visited on 2025-02-05).

[11] National Energy Technology Laboratory. *FE/NETL CO\textsubscript 2 Saline Storage Cost Model: User's Manual*. U.S. Department of Energy, Pittsburgh, PA, 2017.

[12] U.S. Environmental Protection Agency (EPA). Supply, underground injection, and geologic sequestration of carbon dioxide. January 2025. Last modified January 14, 2025. Accessed February 5, 2025. URL: https://www.epa.gov/ghgreporting/supply-underground-injection-and-geologic-sequestration-carbon-dioxide (visited on 2025-02-05).

[13] Ramsey Fahs, Rory Jacobson, Andrew Gilbert, Dan Yawitz, Catherine Clark, Jill Capotosto, Colin Cunliff, Brandon McMurtry, and Uisung Lee. Pathways to commercial liftoff: carbon management. Technical Report, U.S. Department of Energy, 2023. Accessed February 5, 2025. URL: https://liftoff.energy.gov/wp-content/uploads/2024/02/20230424-Liftoff-Carbon-Management-vPUB_update4.pdf (visited on 2025-02-05).

[14] U.S. Congress. H.r.6049 - 110th congress (2007-2008): energy improvement and extension act of 2008. September 2008. URL: https://www.congress.gov/bill/110th-congress/house-bill/6049 (visited on 2025-06-10).

[15] U.S. Congress. H.r.1892 - 115th congress (2017-2018): bipartisan budget act of 2018. February 2018. URL: https://www.congress.gov/bill/115th-congress/house-bill/1892 (visited on 2025-06-10).

[16] U.S. Congress. H.r.5376 - 117th congress (2021-2022): inflation reduction act of 2022. August 2022. URL:

https://www.congress.gov/bill/117th-congress/house-bill/5376 (visited on 2025-06-10).

Model Formulation

This section describes the CCATS optimization model. Please refer to the Glossary for additional details, and to Inputs and Methods for parameter selection.

Objective Function

The core objective of the CCATS optimization model is to minimize the total cost associated with and transporting and storing CO_2 from supply sources to sequestration sites. In the model, each of these locations are represented as nodes, n, while transportation routes or connections between two nodes are represented as arcs, a.

The model is also designed with multiple time periods to accommodate investment in network expansion. A time period is denoted by t.

The objective function (1), encompasses investment including CAPEX and fixed operating and maintenance (O&M) costs (2), variable operating costs including variable O&M and electricity costs (3), and policy-related costs and incentives (4).

$$\min_{\bar{\mathbf{X}}_{\mathbf{a,t}},\ \mathbf{X_{a,t}},\ \mathbf{X_{a,p,t}},\ \mathbf{Q^T_{a,d,t}},\ \mathbf{I^S_{n,t}}} \sum_{t\in\mathcal{T}} \left(C_t^{investment} + C_t^{variable} + C_t^{policy}\right) \tag{1}$$

$$C_t^{investment} = \psi_t^T \sum_{a\in\mathcal{A},\ d\in\mathcal{D}} \left(\theta^T_{a,d,t}\mathbf{Q^T_{a,d,t}}\right) + \psi_t^S \sum_{n\in\mathcal{N}^S} \left(\theta^S_{n,t}\mathbf{I^S_{n,t}}\right) \tag{2}$$

$$C_t^{variable} = \psi_t^{variable}\left(\sum_{a\in\mathcal{A}} \left(\kappa^T_{a,t}\lambda^T_{a,t}X_{a,t}\right) + \sum_{n\in\mathcal{N}^S} \left(\lambda^S_{n,t}\sum_{a\in\mathcal{A}^{in}_n} X_{a,t}\right)\right) \tag{3}$$

$$C_t^{policy} = \psi_t^{policy} \sum_{n\in\mathcal{N}^S,\ a\in\mathcal{A}^{in}_n,\ p\in\mathcal{P}} \zeta_{n,p}\mathbf{X_{a,p,t}} \tag{4}$$

The model solves for several decision variables, both continous and binary. Decision variables are denoted in bold. First, the model determines the flow of CO_2 in arcs, split into three types for tractability: flow that uses existing capacity, $\bar{\mathbf{X}}_{\mathbf{a,t}}$, flow that uses newly constructed capacity, $\mathbf{X_{a,t}}$, and flow that receives different policy incentives p, $\mathbf{X_{a,p,t}}$. Policy incentives are denoted by p.

Second, the model determines the amount of additional pipeline capacity installed, $\mathbf{Q^T_{a,d,t}}$, indexed by diameter size d. Lastly, the model solution includes binary decision variables representing investment in saline storage, $\mathbf{I^S_{n,t}}$. The superscripts T and S denote transportation and storage, respectively.

The parameters in the objective function represent the various costs by type of node. Parameters denoted by ψ represent discount factors which can vary by the type of cost and over time. Parameters denoted by θ represent investment costs, while λ denotes variable costs. Transportation by pipeline requires electricity and is denoted by electricity demand, $\kappa^T_{a,t}$. Policy incentives are denoted by $\zeta_{n,p}$.

Constraints

The model has four groups of constraints: constraints on the volume of flow in each arc, constraints on the balance of CO_2 flowing in and out of the nodes; constraints on transportation investment, and constraints on saline storage investment.

Arc Flow Constraints

$$\bar{\mathbf{X}}_{\mathbf{a,t}} \leq \rho_{a,t}, \quad \forall\, a\in\mathcal{A},\ t\in\mathcal{T} \tag{5}$$

$$\mathbf{X_{a,t}} \leq \sum_{d\in\mathcal{D},t^*<=t-1} \mathbf{Q^T_{a,d,t^*}}, \quad \forall\, a\in\mathcal{A},\ t\in\mathcal{T} \tag{6}$$

$$\mathbf{Q}^{\mathbf{T}}_{\mathbf{a,d,t}} \leq \sigma^{max}_{a,d}, \quad \forall\, a \in \mathcal{A},\ d \in \mathcal{D},\ t \in \mathcal{T} \tag{7}$$

$$X_{a,t} = \bar{\mathbf{X}}_{\mathbf{a,t}} + \mathbf{X}_{\mathbf{a,t}}, \quad \forall\, a \in \mathcal{A},\ t \in \mathcal{T} \tag{8}$$

$$X_{a,t} = \sum_{p \in \mathcal{P}} \mathbf{X}_{\mathbf{a,p,t}}, \quad \forall\, a \in \mathcal{A},\ t \in \mathcal{T} \tag{9}$$

In Equation (5), flow that uses existing pipeline capacity as of the first time period, $\bar{\mathbf{X}}_{\mathbf{a,t}}$, is limited by the existing capacity parameter, $\rho_{a,t}$. This parameter is taken from the data in the early projection years and is determined by the model in later projection years as pipeline capacity is built in NEMS endogenously. Please refer to this section for additional details.

In Equation (6), flow that uses newly constructed capacity, $\mathbf{X}_{\mathbf{a,t}}$, is limited by the total capacity that has been installed in all previous time periods and is available for use in the current time period. This newly constructed capacity is a continuous decision variable that is in turn limited by the parameter $\sigma^{max}_{a,d}$ in Equation (7). This parameter represents the maximum volume that can be installed based on characteristics of available pipelines in the market, particularly diameter.

Constraints (6) and (7) are formulated using the big M method to limit the number of binary decision variables in the model. Otherwise, the optimization problem where a binary investment decision is multiplied by build capacity or flow, which are both decision variables themselves, is a non-linear problem. To avoid this, the investment decision variable, $\mathbf{Q}^{\mathbf{T}}_{\mathbf{a,d,t}}$, is instead defined as a continuous variable. When this variable is zero, meaning no investment is undertaken by the model, the constraints force both capacity and flow to be zero as well.

Equation (8) defines the secondary decision variable, $X_{a,t}$, as the sum of the two flow types for ease of notation. Equation (9) constrains this total to be the sum of CO_2 flow across policies.

Node Flow Balance Constraints

The next set of constraints define the net supply or demand of CO_2 at nodes and ensure that the sum of flows into a node and the sum of flows out of the node must balance out. The subsets of arcs with flow going into and out of a node is denoted by $\mathcal{A}^{in}_n$ and $\mathcal{A}^{out}_n$, respectively.

$$\sum_{a \in \mathcal{A}^{in}_n} X_{a,p,t} - \sum_{a \in \mathcal{A}^{out}_n} X_{a,p,t} = -\phi^{C}_{n,p,t}, \quad \forall\, n \in \mathcal{N}^{C},\ t \in \mathcal{T},\ p \in \mathcal{P} \tag{10}$$

$$\sum_{a \in \mathcal{A}^{in}_n} X_{a,p,t} - \sum_{a \in \mathcal{A}^{out}_n} X_{a,p,t} = 0, \quad \forall\, n \in \mathcal{N}^{TS},\ t \in \mathcal{T},\ p \in \mathcal{P} \tag{11}$$

$$\sum_{a \in \mathcal{A}^{in}_n} X_{a,t} - \sum_{a \in \mathcal{A}^{out}_n} X_{a,t} \leq J_{n,t}, \quad \forall\, n \in \mathcal{N}^{S,saline},\ t \in \mathcal{T} \tag{12}$$

$$\sum_{a \in \mathcal{A}^{in}_n} X_{a,t} - \sum_{a \in \mathcal{A}^{out}_n} X_{a,t} \leq \phi^{S,EOR}_{n,t}, \quad \forall\, n \in \mathcal{N}^{S,EOR},\ t \in \mathcal{T} \tag{13}$$

Equation (10) sets the CO_2 flow balance for capture nodes equal to $-\phi^{C}_{n,p,t}$. This parameter represents the supply of captured CO_2 and is determined by other NEMS modules. Note that this constraint is an equality, so that CCATS must allocate and transport all captured CO_2 received from other NEMS modules.

Equation (11) sets the balance at transshipment points to zero, so that these nodes cannot be used as temporary storage locations.

Equation (12) limits the flow balance in saline storage nodes by the injection parameter $J_{n,t}$. It is defined as an inequality because the storage capacity in geologic formations in the U.S. is an order of magnitude higher than the CO_2 emissions being produced in the data. This parameter determined by the model in later projection years as storage capacity is built in NEMS endogenously. It is further discussed in the section on section_saline_storage.

Equation (13) limits the flow balance to the EOR parameter $\phi^{S,EOR}_{n,t}$. This parameter is determined by HSM. Similar to the constraint on saline storage, it is also defined as an inequality, so the model does not force captured CO₂ to be directed to EOR unless it is economical. As such, there may be EOR demand that is unfulfilled in the model.

Supply of CO₂ from NEMS modules will be specified by policy elgibility, thus the need to index by p in Equations (10). Consequently, the flow balance in the transshipment network also needs to be tracked by policy in Equation (11). However, storage and EOR nodes can accept flow from any policy, thus Equations (12) and (13) are not indexed by p.

Saline Storage Investment Constraints

CCATS represents two aspects of storage: the speed at which CO₂ flow can be stored and the total volume of CO₂ that can be stored.

$$J_{n,t} = \alpha_n + \beta_n \sum_{0 \leq t^\star \leq t-1} \mathbf{I^S_{n,t^\star}}, \quad \forall\, n \in \mathcal{N}^{S,saline},\ t \in \mathcal{T} \tag{14}$$

$$\sum_{a \in \mathcal{A}_n^{in}, 0 \leq t^\star \leq t} \tau_t^\star\, X_{a,t^\star} \leq \gamma_n + \delta_n \sum_{0 \leq t^\star \leq t-1} \mathbf{I^S_{n,t^\star}}, \quad \forall\, n \in \mathcal{N}^{S,saline},\ t \in \mathcal{T} \tag{15}$$

$$\sum_{t \in \mathcal{T}} \mathbf{I^S_{n,t}} \leq \epsilon_n, \quad \forall\, n \in \mathcal{N}^{S,saline} \tag{16}$$

Injectivity, or the rate at which CO₂ can be injected, is defined in Equation (14) and is based on the investment decision in a saline storage node. The parameters α_n and β_n are based on external studies and data.

Cumulative injection, or the total amount of CO₂ that can be injected over all time periods, is defined in Equation (15). The parameters γ_n and δ_n are also based on external studies and data.

Lastly, a storage node is broken down into multiple Areas of Review (AOR) to represent incremental development of a storage location. Equation (16) limits the total number of AORs that can be built at a given node.

Non-Linear MILP Model

The default specification of CCATS is a linear program to save on runtime. However, the model can also be run as a mixed-integer linear program (MILP). In the MILP specification, investment in pipeline capacity is a piecewise-linear function instead of a simple linear function. The following equations are different from the ones described above.

$$\min_{\bar{\mathbf{X}}_{\mathbf{a,t}},\ \mathbf{X}_{\mathbf{a,t}},\ \mathbf{X}_{\mathbf{a,p,t}},\ \mathbf{Q^T_{a,d,t}},\ \mathbf{I^T_{a,d,t}}\ \mathbf{I^S_{n,t}}} \quad \sum_{t \in \mathcal{T}} \left(C_t^{investment} + C_t^{variable} + C_t^{policy} \right) \tag{17}$$

$$C_t^{investment} = \psi_t^T \sum_{a \in \mathcal{A},\ d \in \mathcal{D}} \left(\eta^T_{a,d,t} \mathbf{I^T_{a,d,t}} + \theta^T_{a,d,t} \mathbf{Q^T_{a,d,t}} \right) + \psi_t^S \sum_{n \in \mathcal{N}^S} \left(\theta^S_{n,t} \mathbf{I^S_{n,t}} \right) \tag{18}$$

$$\sum_{d \in \mathcal{D}} \mathbf{I^T_{a,d,t}} \leq 1, \quad \forall\, a \in \mathcal{A},\ t \in \mathcal{T} \tag{19}$$

$$\sigma^{min}_{a,d} \mathbf{I^T_{a,d,t}} \leq \mathbf{Q^T_{a,d,t}}, \quad \forall\, a \in \mathcal{A},\ d \in \mathcal{D},\ t \in \mathcal{T} \tag{20}$$

$$\mathbf{Q^T_{a,d,t}} \leq \sigma^{max}_{a,d} \mathbf{I^T_{a,d,t}}, \quad \forall\, a \in \mathcal{A},\ d \in \mathcal{D},\ t \in \mathcal{T} \tag{21}$$

In the objective function, Equation (1) becomes Equation (17) with an additional binary decision variable $\mathbf{I^T_{a,d,t}}$.

Equation (18) replaces the calculation for investment costs in Equation (2) and includes the piecewise linear function. The coefficient $\eta^T_{a,d,t}$ is as an intercept parameter for each of the segments of the piecewise linear function, while $\theta^T_{a,d,t}$ is a slope parameter for each additional unit of capacity invested in within that segment.

Equation (19) ensures that each arc can only build once per time period, and must be in one of the piecewise linear segments. Equation (20) and Equation (21) enforce the lower and upper bounds per segment of the pipeline piecewise linear function.

Price of Carbon Dioxide

One of the main results of the CCATS optimization model is the price CO_2. Price is an important driver used by other NEMS modules to determine their decision to capture CO_2 and whether to install capture technology.

There are two types of flows in CCATS: flows that are 45Q eligible and flows that are not 45Q eligible. CCATS distinguishes between the two types to enable the model to return two separate prices, one for each 45Q eligibility. This is important, because policy incentives can be a big driver of the decision to provide captured CO_2 to the model.

The price CO_2 comes the duals from the constraints in Equation (10). These equations represent the point at which CCATS receives CO_2 from other NEMS modules, and equates supply to demand. In the optimization, the dual variables reflect the change in the objective function for each unit change in the constraint. Economically, it reflects the present value of the marginal cost to transport, store, and receive policy incentives for an additional unit of captured CO_2.

CCATS uses a highly detailed representation of CO_2 supply facilities, which is more granular than what can be used by other NEMS modules. The constraints need to be aggregated to a census region or census division level to be compatible with NEMS. As such, CCATS calculates the volume-weighted average of the duals from the constraints in a post-processor to aggregate at both the census region and census division levels . In addition, CCATS returns the CO_2 price for the first year of the optimization which aligns with the year that the NEMS model is being run for a particular cycle or iteration.

Relationship between CCATS and NEMS

Within a NEMS run, the CCATS model is executed each model year, iteration, and cycle. For instance, when NEMS runs for the 2025 model year, CCATS optimizes with 2025 as the initial period.

To reduce runtime, CCATS simplifies the temporal resolution of the investment horizon and limits the number of time periods to three. The first time period represents the current NEMS year, the second time period the following year, and the third time represents a longer time horizon to inform long-term decisions. Capacity can be added during any period, but will not available for use until the following time period. CCATS projects operation and capacity expansion of carbon transport and storage over this time horizon.

Table 6. CCATS Time Periods and Capacity Expansion Assumptions

Time Period	Duration (years)	Capacity Expansion
1	1	Yes
2	1	Yes
3	18	Yes

Source: U.S. Energy Information Administration.

Although the model produces solutions for three time periods, only the solution for the first time period is returned to NEMS. For example, a NEMS run for the model year 2025 uses output from the first time period of a CCATS model. Subsequently, when NEMS advances to the 2026 model year, CCATS re-optimizes, now with 2026 as the initial period.

In addition, model parameters are derived from two sources: outputs from other NEMS modules, enabling feedback effects, and expert studies for calibrating remaining parameters. For example, the model also uses NEMS results from 2025 as input parameters for 2026 to allow the model to build capacity over time.

Modeling in Pyomo

The formulation is implemented in Pyomo using the `declare_objective()` and `declare_objective_across_blocks()` methods within the OptimizationModel class.

Constraints are set in the optimization program by `declare_constraints()` and `declare_constraints_across_blocks()`.

Additional documentation for the source code of the CCATS module can be found found in the Model API Reference Section.

Glossary

Superscripts

Superscripts identify the main concepts within CCATS, indicating the relevance of sets, parameters, and variables to those concepts. For clarity and tractability, the number of superscripts is limited. Superscripts are displayed in regular font and listed in Table 7.

Table 7. Superscripts.

Superscript	Short Description
C	Capture
A	Allocation
T	Transport
TS	Transshipment
S	Storage
S, EOR	Storage - Enhanced Oil Recovery
$S, saline$	Storage - Saline aquifer
in	In
out	Out

Sets and subsets

Sets and subsets group model objects with shared characteristics, such as nodes and arcs, enabling efficient application of consistent calculations. Parameters and variables are indexed by these sets and subsets and are displayed in calligraphic font.

Sets and subsets are listed in Table 8. Subsets, denoted by a superscript on the parent set, are listed below their corresponding set. For example, t represents a time period within the set $\mathcal{T}$.

In the code, sets for the optimization program are defined using the `declare_sets()` method within the OptimizationModel class.

Table 8. Sets and subsets.

Set	Subset	Element	CCATS Name	Short Description	Detailed Description
$\mathcal{A}$		a	`s_arcs`	Arcs, connections between nodes	Details
	$\mathcal{A}^{in}$	a	`arcs_in`	Arcs transporting flow into a node	Details
	$\mathcal{A}^{out}$	a	`arcs_out`	Arcs transporting flow away from a node	Details
$\mathcal{D}$		d	`s_transport_options`	Pipeline segment options	Details
$\mathcal{N}$		n	`s_nodes`	Nodes	Details
	$\mathcal{N}^{C}$	n	`s_nodes_supply`	Capture nodes	Details
	$\mathcal{N}^{TS}$	n	`s_nodes_transshipment`	Transport (transshipment) nodes	Details
	$\mathcal{N}^{S}$	n	`s_nodes_demand`	Saline storage or EOR nodes	Details
	$\mathcal{N}^{S,saline}$	n	`s_nodes_demand_storage`	Saline storage nodes	Details
	$\mathcal{N}^{S,EOR}$	n	`s_nodes_demand_eor`	EOR nodes	Details
$\mathcal{P}$		p	`s_policy_options`	CO_2 Policies (for example, 45Q)	Details

Set	Subset	Element	CCATS Name	Short Description	Detailed Description
$\mathcal{T}$		t	`s_time`	Time	Details

Data Source: U.S. Energy Information Administration

Arcs

$\mathcal{A}$ is the set of all arcs a. Arcs represent connections between nodes. Arcs have directionality, therefore a single arc allows flow in one direction. Supply nodes only have the option to build arcs that transport flow away from the supply. Saline storage and EOR nodes only have the option to build arcs that transport flow to storage/EOR. Transshipment nodes have the option to build flow in either direction between nodes. Arcs options are defined in the offline preprocessor. In the optimization model, arcs are indexed by 1) the name (string) of the starting node, and 2) the name (string) of the ending node.

Arcs in

$\mathcal{A}^{in}$ is a subset of $\mathcal{A}$. $\mathcal{A}^{in}$ represents the arcs with flow coming into a node. This subset is used to support the balance of flows coming into a node against the flows leaving a node. $\mathcal{A}^{in}$ is determined in the online preprocessor.

Arcs out

$\mathcal{A}^{out}$ is a subset of $\mathcal{A}$. $\mathcal{A}^{out}$ represents the arcs with flow exiting a node. This subset is used to support the balance of flows coming into a node against the flows leaving a node. $\mathcal{A}^{out}$ is determined in the online preprocessor.

Pipeline options

$\mathcal{D}$ is the set of all pipeline options d. When CCATS is run as a linear program, then each arc only has one pipeline option. When CCATS is run as a mixed integer linear program, it uses a piecewise linear function to select between different pipeline options. The different pipeline options are designed to represent possible combinations of pipeline diameters and number of pumps. Each segment of the piecewise linear is a pipeline option d. $\mathcal{D}$ is determined in the offline preprocessor. Pipeline options are indexed by their name (string).

Nodes

$\mathcal{N}$ is the set of nodes n. Nodes represent locations where pipelines (arcs) either start or end. Nodes include key locations including suppliers of CO_2, EOR demand sites, and saline storage. Nodes are indexed by their name (string).

Supply nodes

$\mathcal{N}^C$ is the subset of nodes where CO_2 capture occurs, thus supplying CO_2 to the network.

Transshipment nodes

$\mathcal{N}^{TS}$ is the subset of nodes where two pipelines join, known as a transshipment node.

Storage nodes

$\mathcal{N}^S$ is the subset of nodes where CO_2 is stored either in EOR or saline storage.

Saline storage nodes

$\mathcal{N}^{S,saline}$ is the subset of nodes $\mathcal{N}^S$ where CO_2 is stored in saline storage.

EOR nodes

$\mathcal{N}^{S,EOR}$ is the subset of nodes $\mathcal{N}^S$ where CO₂ used for Enhanced Oil Recovery (EOR).

Policy options

$\mathcal{P}$ is the set of policy options p. This includes flow that is not eligible for policy incentives and flow that is eligible for 45Q. CCATS does not determine the 45Q eligibility of flow, rather that is provided as an input by NEMS modules supplying CO₂. Policy options are indexed by their name (string).

Time periods

$\mathcal{T}$ is the set of time periods t. CCATS operates with three time periods, which are implemented in Pyomo using a block structure. The first time period represents the current year being analyzed by NEMS. The second and third time period are used to represent the future to support investment decisions. Time periods are indexed by their number (integer).

Parameters

Parameters serve as inputs to the CCATS optimization program and are generally represented using lowercase Greek letters. These parameters are sourced either endogenously from other NEMS modules or exogenously from the offline CCATS preprocessor. We use the notation $\mathbb{R}$ to denote the set of real numbers and $\mathbb{R}_0^+$ to denote the set of non-negative real numbers. Parameters are defined for the optimization program using the `declare_parameters()` method within the OptimizationModel class. A complete list of parameters is provided in Table 9.

Table 9. Parameters.

Parameter	CCATS Name	Data Type	Short Description	Source	Units	Detailed Description
Storage						
α_n	`p_co2_demand_storage`	$\mathbb{R}_0^+$	Existing injectivity	Exogenous	$t\,CO_2/year$	Details
β_n	`p_storage_inj_capacity_adder`	$\mathbb{R}_0^+$	Additional injectivity per AoR	Exogenous	$t\,CO_2/year$	Details
γ_n	`p_storage_injection_net_remaining`	$\mathbb{R}_0^+$	Existing net capacity	Exogenous	$t\,CO_2$	Details
δ_n	`p_storage_injection_adder`	$\mathbb{R}_0^+$	Additional capacity per AoR	Exogenous	$t\,CO_2$	Details
ϵ_n	`p_storage_aors_available`	$\mathbb{R}_0^+$	Number of AoRs available to open	Exogenous	$t\,CO_2$	Details
Policy						
$\zeta_{n,p,t}$	`p_policy_45Q`	$\mathbb{R}$	Policy cost (+) or incentive (-)	Exogenous	$\$/t\,CO_2$	Details
Costs						
$\eta^T_{a,d,t}$	`p_capex_transport_0`	$\mathbb{R}_0^+$	Transport CAPEX - Intercept	Exogenous	$\$$	Details
$\theta^T_{a,d,t}$	`p_capex_transport_slope`	$\mathbb{R}_0^+$	Transport CAPEX	Exogenous	$\$/t\,CO_2$	Details

Parameter	CCATS Name	Data Type	Short Description	Source	Units	Detailed Description
$\theta^S_{n,t}$	`p_capex_storage`	$\mathbb{R}^+_0$	Storage CAPEX	Exogenous	$\$/t\,CO_2$	Details
$\kappa^T_{a,t}$	`p_electricity_demand`	$\mathbb{R}^+_0$	Transport electricity consumption	Exogenous	$MWh/t\,CO_2$	Details
$\lambda^T_{a,t}$	`p_opex_transport`	$\mathbb{R}^+_0$	Transport electricity cost	Exogenous	$\$/MWh$	Details
$\lambda^S_{n,t}$	`p_opex_storage`	$\mathbb{R}^+_0$	Storage OPEX	Exogenous	$\$/t\,CO_2$	Details
Transport						
ρ_a	`p_transport_capacity_existing`	$\mathbb{R}^+_0$	Exisiting capacity	Exogenous	$t\,CO_2/year$	Details
$\sigma_{a,d}$	`p_transport_capacity_adder`	$\mathbb{R}^+_0$	Maximum capacity per new build	Exogenous	$t\,CO_2/year$	Details
Net Supply						
$\phi^C_{n,p}$	`p_co2_supply`	$\mathbb{R}^+_0$	CO_2 captured at source	Endogenous	$t\,CO_2/year$	Details
$\phi^{S,EOR}_n$	`p_co2_demand_eor`	$\mathbb{R}^+_0$	CO_2 demand for EOR	Endogenous	$t\,CO_2/year$	Details
Financing/discounting and time						
τ_t	`p_duration`	$\mathbb{R}^+_0$	Duration of time period	Exogenous	$years$	Details
ψ^T_t	`p_discount_invest_storage`	$\mathbb{R}^+_0$	Discount factor for storage investment	Endogenous	—	Details
ψ^S_t	`p_discount_invest_transport`	$\mathbb{R}^+_0$	Discount factor for transport investment	Endogenous	—	Details
$\psi^{variable}_t$	`p_discount_variable`	$\mathbb{R}^+_0$	Discount factor for variable costs	Endogenous	—	Details
ψ^{policy}_t	`p_discount_policy`	$\mathbb{R}^+_0$	Discount factor for policy costs	Endogenous	—	Details

Existing injectivity

α_n is the injectivity (amount of CO_2 that can be injected per year) at node n before the current model year.

Additional injectivity

β_n is the additional injectivity that is added to node n per additional AoR opened.

Existing net capacity

γ_n is the injection capacity remaining at node n before the current model year.

Additional capacity

δ_n is the additional injection capaicty added to node n per additional AoR opened.

Available AoRs

ϵ_n is the number of AoRs available to open at node n.

Policy cost

$\zeta_{n,p,t}$ is the cost of policy p. Incentives such as 45Q are provided as a negative cost.

Transport investment - intercept

$\eta^T_{a,d,t}$ is the intercept for transport investment costs. This is only used in the non-linear version of CCATS.

Transport investment - slope

$\theta^T_{a,d,t}$ is the slope of transport investment costs.

Storage investment

$\theta^S_{n,t}$ is the investment cost of storage.

Transport electricity consumption

$\kappa^T_{a,t}$ is the amount of electricity consumed to transport a tonne of CO_2.

Transport electricity cost

$\lambda^T_{a,t}$ is the cost per unit of electricity consumed for arc a.

Storage variable cost

$\lambda^S_{n,t}$ is the variable cost of storage.

Existing transport capacity

ρ_a is the capacity of transport arcs before the current model year.

Transport capacity adder

$\sigma_{a,d}$ is the limit for adding capacity to an arc for the current pipeline option d.

CO_2 supply

$\phi^C_{n,p}$ is the amount of CO_2 supplied at node n and of policy p.

EOR demand

$\phi_n^{S,EOR}$ is the maximum CO₂ demand for EOR at node n.

Duration

τ_t is the duration of time period t.

Transport discount factor

ψ_t^T is a multiplier to finance and discount payments for transportation investments made in time period t. It is calculated by `calculate_discount_investment()`.

$$\psi_t^T = \left(\frac{RINT_t(1+RINT_t)^n}{(1+RINT_t)^n - 1} * \sum_{y=y_t+1}^{y_t+1+n} DIS_{y_0,y} \right) + \left(FOM * \sum_{y=y_t+1}^{y_t+1+n} INF_{y_t,y} * DIS_{y_0,y} \right)$$

with the discount factor, DIS, and inflation factor, INF, from year y_0 to y_1 defined as:

$$DIS_{y_0,y_1} = \begin{cases} 1 & \forall\, y_1 - y_0 = 0 \\ \prod_{y^\star=y_0}^{y_1} \frac{1}{(1+RDIS_{y^\star})} & \forall\, y_1 - y_0 > 0 \end{cases}$$

$$INF_{y_0,y_1} = \begin{cases} 1 & \forall\, y_1 - y_0 = 0 \\ \prod_{y^\star=y_0}^{y_1} (1+RINF_{y^\star}) & \forall\, y_1 - y_0 > 0 \end{cases}$$

The discount and inflation factor is summed starting at $y_t + 1$ because the first payment is assumed to occur one year after the investment decision.

These equations rely on eight inputs:

- time inputs:
  - t is the time period of the investment decision,
  - y_0 is the NEMS year that time period 0 begins,
  - y_t is the NEMS year that time period t begins,
  - n are the number of years that CAPEX is financed for transport and storage, respectively,
- financing inputs:
  - FOM is the fraction of CAPEX paid each year as Fixed O&M for transport and storage, respectively,
  - $RINF_y$ is the inflation rate of year y,
  - $RDIS_y$ is the discount rate of year y equal to the real rate + inflation rate,
  - $RINT_t$ is the interest rate used for borrowing in time period t, equal to the real rate + inflation rate + risk premia.

Storage discount factor

ψ_t^S is a multiplier to finance and discount payments for storage investments made in time period t. It is calculated by `calculate_discount_investment()`.

$$\psi_t^S = \left(\frac{RINT_t(1+RINT_t)^n}{(1+RINT_t)^n - 1} * \sum_{y=y_t+1}^{y_t+1+n} DIS_{y_0,y} \right) + \left(FOM * \sum_{y=y_t+1}^{y_t+1+n} INF_{y_t,y} * DIS_{y_0,y} \right)$$

Variable cost discount factor

$\psi_t^{variable}$ is a multiplier to discount variable costs occurring in time period t. Variable costs are input for a single year, so $\psi_t^{variable}$ also accounts for repeated costs in multi year time periods. It is calculated by `calculate_discount_variable_policy()`.

$$\psi_t^{variable} = \sum_{y=y_t}^{y_t+\Delta_t} \left(DIS_{y_0,y} \right)$$

where Δ_t is the duration of the time period t.

Policy cost discount factor

ψ_t^{policy} is a multiplier to discount policy costs occurring in time period t. Policy costs are input for a single year, so ψ_t^{policy} also accounts for repeated costs in multi year time periods. It is calculated by `calculate_discount_variable_policy()`.

$$\psi_t^{policy} = \sum_{y=y_t}^{y_t+\Delta_t} \left(DIS_{y_0,y} \right)$$

Variables

Unknowns (decisions) to be solved by the mathematical program. They are split into primary and secondary decision variables. All variables use the uppercase Roman alphabet, with subscripts in lowercase. Primary decision variables are shown in **bold**. Secondary decisions variables are dependent on primary decision variables. Variables are shown in the order they are declared in the code. We use the symbol $\mathbb{R}$ to denote the set of real numbers, $\mathbb{R}_0^+$ to denote non-negative real numbers, and $\mathbb{B}$ to denote binary variables. Variables are set-up for the optimization program by `declare_variables()`. Variables are listed in Table 10.

Table 10. Variables.

Variable	CCATS Name	Data Type	Short Description	Units	Detailed Description
	Costs				
$C_t^{investment}$	`e_sum_costs_investment`	$\mathbb{R}_0^+$	Investment costs	\$	Details
C_t^{policy}	`e_sum_costs_policy`	$\mathbb{R}$	Policy costs	\$	Details
$C_t^{variable}$	`e_sum_costs_variable`	$\mathbb{R}_0^+$	Variable costs	\$	Details
	Investment				
$\mathbf{I_{a,d,t}^T}$	`vb_transport_investment`	$\mathbb{B}$	Transport investment decision	0 or 1	Details
$\mathbf{I_{n,t}^S}$	`v_storage_investment`	$\mathbb{R}_0^+$	Storage investment decision	# of AoRs	Details
	Storage - injectivity				
$J_{n,t}$	`v_storage_inj_capacity`	$\mathbb{R}_0^+$	Storage injection capacity	$t\,CO_2/year$	Details
	Pipeline Capacity				
$\mathbf{Q_{a,d,t}}$	`v_transport_capacity_added`	$\mathbb{R}_0^+$	Transport capacity constructed in the current time step	$t\,CO_2/year$	Details
	Pipeline Flow				
$X_{a,t}$	`v_flow`	$\mathbb{R}_0^+$	Flow of arc	$t\,CO_2/year$	Details
$\mathbf{\bar{X}_{a,t}}$	`v_flow_base`	$\mathbb{R}_0^+$	Flow using existing capacity (built before CCATS run)	$t\,CO_2/year$	Details
$\mathbf{X_{a,t}}$	`v_flow_add`	$\mathbb{R}_0^+$	Flow using new capacity (built during CCATS run)	$t\,CO_2/year$	Details
$\mathbf{X_{a,p,t}}$	`v_flow_by_policy`	$\mathbb{R}_0^+$	Flow by policy	$t\,CO_2/year$	Details

Investment costs

$C_t^{investment}$ is the sum of investment costs committed in time period t.

Policy costs

C_t^{policy} is the sum of policy costs occurring during time period t.

Variable costs

$C_t^{variable}$ is the sum of variable costs occurring during time period t.

Transport investment

$\mathbf{I^T_{a,d,t}}$ is the transport investment decision at node n in time period t. This is only used in the non-linear (MILP) version of CCATS. This variable is in bold to indicate that it is a primary decision variable.

Storage investment

$\mathbf{I^S_{n,t}}$ is the storage investment decision at node n in time period t. This variable is in bold to indicate that it is a primary decision variable.

Injectivity

$J_{n,t}$ is the available injectivity of node n in time period t.

Transport capacity added

$\mathbf{Q_{a,d,t}}$ is the amount of capacity added to arc a of type d in time period t. This variable is in bold to indicate that it is a primary decision variable.

Total flow

$X_{a,t}$ is the total flow moving through arc a in time period t.

Existing capacity flow

$\mathbf{\bar{X}_{a,t}}$ is the flow using existing transport capacity through arc a in time period t. This variable is in bold to indicate that it is a primary decision variable.

New capacity flow

$\mathbf{X_{a,t}}$ is the flow using new transport capacity through arc a in time period t. This variable is in bold to indicate that it is a primary decision variable.

Flow by policy

$\mathbf{X_{a,p,t}}$ is the flow through arc a in time period t and indexed by policy p. This variable is in bold to indicate that it is a primary decision variable.

How to Run CCATS

CCATS can be run either standalone in a python IDE, or within the NEMS integrated model framework. The CCATS module is written in Python and Pyomo. Documentation for the source code of the CCATS module can be found found in the Model API Reference Section.

Running CCATS Standalone

When CCATS is run standalone, CO_2 supply and demand volumes are exogenous.

To run CCATS standalone:

1. Open the CCATS model directory in a Python IDE (Pycharm, Microsoft Visual Studio, Spyder, etc.)
2. Create and assign a python interpreter including all the Python libraries listed in CCATS Dependencies.
3. Select CCATS run options from the *setup* files stored in the CCATS *input* directories (i.e. model solver, debug outputs, etc.)
4. Run CCATS from **ccats.py**.
5. Review results in the CCATS *debug* directories.

Running CCATS in NEMS

1. Select CCATS run options from the *setup* files stored in the CCATS *input* directories (i.e. model solver, debug outputs, etc.)
2. Select NEMS run options from the *scedes* files store in the NEMS *scedes* directory
3. Run the *Run_NEMS.bat* file and select a run repository and scedes file.
4. Review results in NEMS report writer and CCATS *debug* directories.

CCATS Dependencies

CCATS relies on the below list of Python libraries to run.

- Libraries included with the default distribution or available via pip or conda:
    - sys
    - os
    - io
    - shutil
    - pathlib
    - logging
    - argparse
    - shutil
    - pylint
    - tabulate
    - pylab
    - itertools
    - warnings
    - pickle
    - numpy
    - pandas
    - pyomo
    - matplotlib
    - folium (for mapping)
    - xpress (if using the FICO Xpress solver)
- NEMS specific libraries:
    - pyfiler1 - maintined by the NEMS Integration Team.

Inputs and Methods

Inputs to CCATS are contained in .CSV files. These inputs determine high level assumptions and how the primary CCATS modules and submodules operate. They also include switches for various outputs or features of the model.

Module inputs

Table 11 includes the inputs that are used by `Module`.

Table 11. General inputs

parameter	value	type	description
threads	4	int	number of threads to use (Python code + optimization)
hist_setup	hist_setup.csv	input_file	Historical Data setup file
fin_setup	fin_setup.csv	input_file	Financial Assumptions setup file
preproc_setup	preproc_setup.csv	input_file	Preprocessor setup file
opt_setup	opt_setup.csv	input_file	Optimization setup file
postproc_setup	postproc_setup.csv	input_file	Postprocessor setup file
restart_in	restart_CCATSin.unf	input_file	Restart file input for local standalone CCATS runs
restart_out	restart_CCATSo.unf	output_file	Restart file output by CCATS
preproc_switch	TRUE	switch	Run CCATS Preprocessor if True
ccats_opt_switch	TRUE	switch	Run CCATS Optimization if True
postproc_switch	TRUE	switch	Run CCATS Postprocessor if True
output_switch	TRUE	switch	Run CCATS Output Processor if True
update_45q_switch	TRUE	switch	Update the 45Q policy path with input from setup.txt “eor_45q” and “saline_45q” variables if True
debug_switch	FALSE	switch	Output debug files if True (will result in longer model runtime)
debug_restart_itr_switch	FALSE	switch	Output the restart file debug file every iteration if True (will result in slightly longer model runtime)
write_restart_before_run_switch	FALSE	switch	Output the restart file debug received by CCATS before running if True
output_restart_unf_switch	FALSE	switch	Output the restart file as a .unf after running CCATS if True
visualize_preproc_switch	FALSE	switch	Output folium visualizations of optimization inputs if True
visualize_postproc_switch	FALSE	switch	Output folium visualizations of optimization outputs if True
pytest_switch	FALSE	switch	Perform pytest internal validation tests if True

parameter	value	type	description
price_reset_switch	FALSE	switch	Reset CO2 prices to produce endogenous CO2 prices from models which would otherwise have 0 CO2 capture if True
price_reset_value_45q	-35	input_variable	CO2 price reset value when “price_reset_switch” == True
price_reset_value_ntc	-9.52	input_variable	CO2 price reset value when “price_reset_switch” == True
price_average_switch	TRUE	switch	CO2 prices are set as a weighted average of duals by CD if True
price_marginal_switch	FALSE	switch	CO2 prices are set as the marginal CO2 price by CD if True
linear_model_switch	TRUE	switch	Model is run as a linear model (vs. a MIP) if True
mapping	mapping.csv	input_file	Regionl NEMS mapping (i.e. census divisions to census regions)
co2_eor_mapping	co2_eor_mapping.csv	input_file	Mapping of CO2 EOR plays to census divisions/census regions
idm_mapping	idm_mapping.csv	input_file	Mapping of IDM CO2 capture volumes from cenesus regions to census divisions based on historical production ratios
co2_supply_index	co2_supply_index.csv	input_file	Mapping of CO2 capture facility types to NEMS Restart variable index
co2_seq_index	co2_seq_index.csv	input_file	Mapping of CO2 sequestration types to NEMS Restart variable index
year_start	2024	input_variable	First model run year
year_aeo	2025	input_variable	AEO year
year_final	2050	input_variable	Final model run year
year_base_price	1987	input_variable	Base $ year
eor_45q	60	input_variable	Current 45Q tax credit value for CO2 EOR
saline_45q	85	input_variable	Current 45Q tax credit value for saline formation storage
year_new_45q	2022	input_variable	Year current 45Q tax credit policy took effect
year_45q_duration	12	45Q tax credit policy duration	years
year_45q_last_new	2038	Final year for which new 45Q-eligible facilities can be retrofit/built	year

parameter	value	type	description
legacy_eor_45q	35	Legacy 45Q tax credit value for CO2 EOR	
legacy_saline_45q	50	Legacy 45Q tax credit value for saline formation storage	
year_leg_45q	2017	Year legacy tax credit value took effect	
supply_select_ts_penalty	1	input_variable	Penalty for ts node costs in the model

Financial inputs

Table 12 shows the inputs that are used by `CCATS_Finance`.

Table 12. Financial assumptions

parameter	value	type	description	units
debt_ratio	0.4	input_variable	Ratio of debt financing	fraction
return_over_capital_cost	0.05	input_variable	Assumed required return over capital cost for investment	fraction
fed_tax_rate	0.21	input_variable	Assumed federal tax rate	rate
duration_block0	1	input_variable	Duration of optimization model block 0	years
duration_block1	1	input_variable	Duration of optimization model block 1	years
duration_block2	18	input_variable	Duration of optimization model block 2	years
duration_45q	12	input_variable	Duration of 45Q tax credit policy for a single facility	years
financing_risk_premia	0.02	input_variable	Assumed risk premia required by investors over risk-free market rate	fraction
financing_years_transport	20	input_variable	Assumed financing years for pipeline transport investments	years
financing_years_storage	26	input_variable	Assumed financing years for saline formation storage investments	years
fixed_om_fraction_transport	0.025	input_variable	Fixed O&M costs as a percentage of capital costs for pipeline transport	fraction
fixed_om_fraction_storage	0.087	input_variable	Fixed O&M costs as a percentage of capital costs for saline formation storage	fraction
site_dev_years_storage	4	input_variable	Assumed number of years for storage site development	years
construction_years_storage	2	input_variable	Assumed number of years for storage site construction	years

Preprocessor inputs

Table 13 shows the inputs that are used by `Preprocessor`.

Table 13. Preprocessor inputs

parameter	value	type	description	units
storage_formations_lookup	storage\storage_formations_lookup.csv	input_file	Lookup of saline formation attributes	

parameter	value	type	description	units
co2_supply_facilities	co2_facilities\co2_facility_lookup.csv	input_file	input_file	Lookup of CO2 capture facility attributes
pipeline_lookup_multi	transport\master_pipeline_lookup_multi.csv	input_file	Lookup of CO2 pipeline transportation attributes	
ts_multiplier	transport\ts_multiplier.csv	input_file	Set of CO2 multipliers for TS arcs by year	
hsm_eor_centroid	demand\hsm_eor_centroid.csv	input_file	CO2 demand from HSM centroids during STEO years	
small_sample_switch	FALSE	switch	Model solves for only a single census division for testing if TRUE	
small_sample_division	7	input_variable	Census division solved with “small_sample_switch” == TRUE	
split_45q_supply_switch	TRUE	switch	Split CO2 supply into 45Q and NTC components if TRUE	
block_45q_yr	6	input_variable	Decision variable for number of 45Q eligibility years remaining in model (see method: determine_facility_eligibility_45q)	years
new_pipes_to_exist_facilities	2026	First year that new pipelines can be built to existing capture facilities (first operational in the following year)	years	
tech_learn_ammonia	0.007984	tech_rate	Tech rate for capture at ammonia facilities	fr/yr
tech_learn_ethanol	0.00	tech_rate	Tech rate for capture at ethanol facilities	fr/yr
tech_learn_cement	0.01	tech_rate	Tech rate for capture at cement facilities	fr/yr
tech_learn_ng_processing	0.01	tech_rate	Tech rate for capture at natural gas processing	fr/yr
tech_learn_other	0.01	tech_rate	Tech rate for capture at all other facilities	fr/yr
tech_learn_power	0.00	tech_rate	Tech rate for capture at power plants (coal and natural gas)	fr/yr
tech_learn_transport	0.01	tech_rate	Tech rate for carbon transport	fr/yr
tech_learn_storage	0.01	tech_rate	Tech rate for carbon storage	fr/yr
transport_buffer	1.15	Buffer capacity for new pipeline builds in the optimization	input_variable	
storage_buffer	1.15	Buffer capacity for new saline formation storage	input_variable	

parameter	value	type	description	units
		builds in the optimization		

Optimization model

Table 14 includes the inputs that are used by `OptimizationModel`.

Table 14. Optimization model inputs

parameter	value	type	description
opt_check_solver_status	FALSE	switch	Check solver status and attempt to troubleshoot cause for solve fail if TRUE
troubleshoot_datatypes	FALSE	switch	Log datatypes to support troubleshooting if TRUE
debug_opt_log	FALSE	switch	Output solver output log if TRUE
use_solver_xpress_persistent	TRUE	bool	Use the xpress persistent solver (NEMS default - note: the first solver selected is used)
use_solver_xpress_direct	FALSE	bool	Use the xpress direct solver (note: the first solver selected is used)
use_solver_xpress	FALSE	bool	Use the default version of the xpress solver (note: the first solver selected is used)
use_solver_highs	FALSE	bool	Use the HiGHS solver (note: the first solver selected is used)
xpress_path	C:/xpress8_12/bin/xpauth.xpr	str	path to xpress' xpauth.xpr license file
soltimelimit	600	int	solution time limit per iteration (seconds)
maxmemorysoft	16000	int	MB available for solve

Postprocessor inputs

Table 15 shows the inputs that are used by `Postprocessor`.

Table 15. Postprocessor inputs

parameter	value	type	description	units
price_limit	1000	input_variable	Maximum price (+/-) returned by CCATS	$1987
price_peg	0.3	input_variable	Maximum price (+/-) movement allowed per cycle relative to the previous cycle	$1987
slack_threshold	1	input_variable	Threshold at which slack infeasibilities are recorded in debug outputs	tonnes
log_b1_infeasibilities	TRUE	switch	Log b1 infeasibilties if TRUE	
log_b2_infeasibilities	FALSE	switch	Log b2 infeasibilties if TRUE	
infeasibility_threshold_pct	0.5	input_variable	Log infeasibilities if greater than provided percentage of total flow	%

.

ccats module

Main execution function for the Carbon Capture, Transportation, Allocation and Sequestration (CCATS) Module.

CCATS Main: Summary

ccats.py is the main execution function for CCATS. It is called from NEMS **main.py**, performs basic model setup functions, and then calls Module: Summary to run the main CCATS processes. The file operates as follows:

1. **ccats.py** is called from NEMS **main.py** or run directly from an IDE.
2. Logging setup:
   a. The previous run's log is deleted from the directory.
   b. Logger is configured and written to *"ccats_debug.log"*.
3. File directory paths are instantiated.
4. `get_args()` is called, pulling run information from the NEMS command line for integrated runs (i.e. iteration and model year).
5. `run_ccats()` is called, kicking off the main CCATS run operations:
   a. `Module` in Module: Summary is declared, main directories are declared, and main setup file is identified.
   b. `module.Module.setup()` in Module: Summary is called to perform year 1 model setup.
   c. CCATS determines whether to run standalone vs. integrated based on run location.
   d. Main CCATS functions are run via Module: Summary.
   e. After main CCATS model functions run, results are prepared and reported to the restart file in `restart.Restart.write_results()`.

CCATS Main: Input Files

None

CCATS Main: Functions and Class Methods

- `get_args()` - Parses the arguments passed in via command line of NEMS
- `run_ccats()` - Executes main CCATS processes from Module: Summary.

CCATS Main: Output Debug Files

None

CCATS Main: Output Restart Variables

None

CCATS Main: Code

`ccats.`**`get_args`**`(`*`in_args=None`*`)` [source]

Parses the arguments passed in via command line of NEMS. NOTE: NEMS calls Pyfiler with trailing options for PyFiler to parse through for the input file, output file, and direction of restart to hdf formatting.

Parameters: **in_args** (*list*) – in_args first set to none, then creates a list of system arguments passed via the command line.

Returns: **arguments** – command line arguments saved into namespace variable.

Return type: Namespace

`ccats.`**`run_ccats`**`(`*`year, iteration, pyfiler1, cycle, scedes`*`)` [source]

Executes main CCATS processes from Module: Summary.

- Instantiates main directory,
- Declares `module.Module` parent class for CCATS,
- Runs `module.Module.setup()` processes from Module: Summary,
- Runs `module.Module.run()` processes from Module: Summary, and
- Runs `restart.Restart.write_results()` processes from Restart: Summary.

Parameters:

- **year** (*str or int*) – Current run year.
- **iteration** (*str or int*) – Current NEMS iteration.
- **pyfiler1** (*None or pyfiler*) – Tool for reading and writing to the restart file.
- **cycle** (*str or int*) – Current NEMS cycle.
- **scedes** (None or scedes file inputs from **main.py**.) – NEMS Scenario Description data.

Return type: None

ccats_common package

Submodules

ccats_common.common module

Utility file containing miscellaneous common functions used in CCATS.

Common: Summary

This is a utility file containing general functions that are either a) used frequently in a single submodule, or are used by several submodules. Example functions include `read_dataframe()`, which is a universal function for reading files into dataframes, and `calculate_inflation()` which applies the NEMS inflation multipliers to CCATS DataFrames.

Convention for import alias is import common as com

Common: Input Files

Unique to each function.

Model Functions and Class Methods

- `read_dataframe()` - Reads multiple filetypes into python as Pandas DataFrames, checking for nans.
- `calculate_inflation()` - Inflation calculator using restart file inflation multiplier, applied to Pandas DataFrames.
- `array_to_df()` - Creates a Pandas DataFrame from a NumPy array.
- `df_to_array()` - Creates a NumPy array from a Pandas DataFrame.
- `unpack_pyomo()` - Unpacks Pyomo results and converts results to DataFrames.
- `check_results()` - Checks optimization termination condition.
- `align_index()` - Aligns the index types of two tables based on the restart variable type.
- `compare_lists()` - Check if list A is equal or a subset of list B.
- `check_dicts_ruleset()` - Check the list of dicts against defined ruleset.

Common: Output Debug Files

None

Common: Output Restart Variables

None

Common: Code

ccats_common.common.**read_dataframe**(*filename*, *sheet_name=0*, *index_col=None*, *skiprows=None*, *to_int=True*) [source]

Reads multiple filetypes into python as pandas DataFrames, checking for nans.

Parameters:
- **filename** (*str*) – Filename, including file type extension (e.g. .csv).
- **sheet_name** (*str*) – Name of the sheet if using excel or hdf.
- **index_col** (*int, str, list, or False*) – Column number to use as row labels.
- **skiprows** (*int*) – Column number to use as row labels.
- **to_int** (*bool*) – If True, values are converted to integers.

Return type: DataFrame

ccats_common.common.**calculate_inflation**(*rest_mc_jpgdp, from_year, to_year=None*) [source]

Inflation calculator using restart file inflation multiplier, applied to pandas DataFrames.

Parameters:
- **rest_mc_jpgdp** (*DataFrame*) – DataFrame of NEMS restart file inflation multipliers indexed by year.
- **from_year** (*int*) – Starting year for inflation calculation.
- **to_year** (*None or int*) – Target year for inflation calculation, if None defaults to 1987.

Return type: float

ccats_common.common.**array_to_df**(*array*) [source]

Create DataFrame from NumPy array.

- Creates a multi-index DataFrame from a multi-dimensional NumPy array
- Each array dimension is stored as a binary index column in the DataFrame multi-index
- Multi-index columns are currently numbered to reflect array dimension level

Parameters: **array** (*NumPy array*) – A multi-dimensional array containing restart file data

Return type: DataFrame

ccats_common.common.**df_to_array**(*df*) [source]

Turns multi-index DataFrame into multi-dimensional NumPy array.

- Creates a multi-dimensional array from a multi-index dataframe
- Each DataFrame multi-index is stored as an array dimension

Parameters: **df** (*DataFrame*) – A multi-index DataFrame containing restart file data.

Return type: NumPy array

ccats_common.common.**unpack_pyomo**(*variable, values, levels*) [source]

Tool for unpacking Pyomo variable outputs and converting them to DataFrames.

Parameters:
- **variable** (*Pyomo Var*) – Target variable to unpack.
- **values** (*Pyomo Datatype (for example, pyo.value)*) – Pyomo value type.
- **levels** (*int*) – Number of levels in array to unpack (i.e. CO$_2$ supplied from i is one level, while CO:sub:`2 piped from i to j is two levels).

Return type: df

ccats_common.common.**check_results**(*results, SolutionStatus, TerminationCondition*) [source]

Check Pyomo termination condition

Parameters:
- **results** (*pyomo.SolverResults*) – Results from Pyomo.
- **SolutionStatus** (*enum.EnumType*) – Solution Status from Pyomo.
- **TerminationCondition** (*enum.EnumType*) – Termination Condition from Pyomo.

Returns: True = problem with solution, False = good solution.

Return type: bool

`ccats_common.common.`**`align_index`**`(`*`df1, df2, restart_var=None`*`)` [source]

Aligns the index types of two tables based on the restart variable type.

- If restart_var is provided we use that to determine the index types
- If restart_var is none, we assume that df1 has the correct index type

Parameters:
- **df1** (*DataFrame (2 dimensional)*) – Restart variable table.
- **df2** (*DataFrame (2 dimensional)*) – Restart variable table.
- **restart_var** (*DataFrame (2 dimensional)*) – Restart variable, used to decide the index type.

Returns:
- **df1** (*DataFrame (2 dimensional)*) – Updated version of df1 parameter.
- **df2** (*DataFrame (2 dimensional)*) – Updated version of df2 parameter.

`ccats_common.common.`**`compare_lists`**`(`*`lista, listb`*`)` [source]

Check if lista is equal or a subset of listb

- used to check that node ids(supply, hubs, storage)in lista

  are part of the set in listb

Parameters:
- **lista** (*list*) – First list for comparison.
- **listb** (*list*) – Second list for comparison.

Return type: None

`ccats_common.common.`**`check_dicts_ruleset`**`(`*`dict_list`*`)` [source]

Check the list of dicts against defined ruleset

- 1. check for nans
- 2. make sure lengths of each dict are the same length

Parameters: **dict_list** (*list*) – list of dicts

Return type: None

ccats_common.common_debug module

Utilities for debugging CCATS.

Common Debug: Summary

This file contains functions to:

1. Perform code analysis with pylint.
2. Debug DataFrames.
3. Debug Pyomo models.

Common Debug: Input Files

Unique to each function.

Common Debug: Model Functions and Class Methods

- `run_pylint()` - Runs pylint to analyze code.
- `check_nans()` - Checks a DataFrame for NaNs.

- `check_infvalues()` - Checks a DataFrame for infinity values.
- `check_index_type()` - Checks two DataFrame to see if they are aligned for a .merge() or .update()
- `print_table()` - Prints a table for easy debugging.
- `compute_infeasibility_explanation()` - used to debug Pyomo models (from Pyomo development team).

Common Debug: Output Debug Files

None

Common Debug: Output Restart Variables

None

Common Debug: Code

`ccats_common.common_debug.`**`run_pylint`**`(`*`filelist=None, errors_only=False`*`)` [source]

Runs pylint.

Parameters:
- **filelist** (*list*) – List of files to evalute, if None, then pylint will evalute the cwd
- **errors_only** (*bool*) – Determines if we run pylint against all checks or errors only

Return type: bool

`ccats_common.common_debug.`**`check_nans`**`(`*`df, tablename='my_table'`*`)` [source]

Checks table (DataFrame) for nan values.

Parameters:
- **df** (*DataFrame (2 dimensional)*) – Restart variable table
- **tablename** (*string*) – Name for the output file

Return type: bool

`ccats_common.common_debug.`**`check_infvalues`**`(`*`df, tablename='my_table'`*`)` [source]

Checks DataFrame for infinity values.

Parameters:
- **df** (*DataFrame (2 dimensional)*) – Restart variable table.
- **tablename** (*str*) – Name for the output file.

Return type: bool

`ccats_common.common_debug.`**`check_index_type`**`(`*`df1, df2`*`)` [source]

Checks two tables (DataFrames) index types to see if they are aligned for a .merge() or .update().

- Checks if index type of df1 is equal to index type of df2
- Logs the result

Parameters:
- **df1** (*DataFrame (2 dimensional)*) – Restart variable table.
- **df2** (*DataFrame (2 dimensional)*) – Restart variable table.

Return type: bool

`ccats_common.common_debug.`**`print_table`**`(`*`df, outputfilename=None`*`)` [source]

Prints a table for easy debugging.

Parameters:
- **df** (*DataFrame (2 dimensional)*) – Restart variable table
- **outputfilename** (*str*) – Name of output file

Return type: None

class ccats_common.common_debug.**_VariableBoundsAsConstraints** [source]

Bases: `object`

PYOMO DEBUG FUNCTION FROM PYOMO DEVELOPMENT TEAM

Replace all variables bounds and domain information with constraints. Leaves fixed Vars untouched (for now)

pyo *= <module 'pyomo.environ' from 'C:\\python_environments\\aeo2025_py311_doc\\Lib\\site-packages\\pyomo\\environ\\__init__.py'>*

class **AddSlackVariables**(***kwds*)

Bases: `NonIsomorphicTransformation`

This plugin adds slack variables to every constraint or to the constraints specified in targets.

CONFIG *= <pyomo.common.config.ConfigDict object>*

__init__(***kwds*)

_apply_to(*instance, **kwds*)

_apply_to_impl(*instance, **kwds*)

class **IsomorphicTransformation**(***kwds*)

Bases: `Transformation`

Base class for 'lossless' transformations for which a bijective mapping between optimal variable values and the optimal cost exists.

__init__(***kwds*)

unique_component_name(*name*)

class **ComponentMap**(**args, **kwds*)

Bases: `Mixin`, `MutableMapping`

This class is a replacement for dict that allows Pyomo modeling components to be used as entry keys. The underlying mapping is based on the Python id() of the object, which gets around the problem of hashing subclasses of NumericValue. This class is meant for creating mappings from Pyomo components to values. The use of non-Pyomo components as entry keys should be avoided.

A reference to the object is kept around as long as it has a corresponding entry in the container, so there is no need to worry about id() clashes.

We also override __setstate__ so that we can rebuild the container based on possibly updated object ids after a deepcopy or pickle.

*** An instance of this class should never be deepcopied/pickled unless it is done so along with the components for which it contains map entries (e.g., as part of a block). ***

_dict

__init__(**args, **kwds*)

_abc_impl *= <_abc._abc_data object>*

clear() → None. Remove all items from D.

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

setdefault(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

class **ComponentSet**(**args*)

Bases: `MutableSet`

This class is a replacement for set that allows Pyomo modeling components to be used as entries. The underlying hash is based on the Python id() of the object, which gets around the problem of hashing subclasses of NumericValue. This class is meant for creating sets of Pyomo components. The use of non-Pyomo components as entries should be avoided (as the behavior is undefined).

References to objects are kept around as long as they are entries in the container, so there is no need to worry about id() clashes.

We also override __setstate__ so that we can rebuild the container based on possibly updated object ids after a deepcopy or pickle.

*** An instance of this class should never be deepcopied/pickled unless it is done so along with its component entries (e.g., as part of a block). ***

_data

__init__(**args*)

_abc_impl = *<_abc._abc_data object>*

add(*val*)

Add an element.

clear()

Remove all elements from this set.

discard(*val*)

Remove an element. Do not raise an exception if absent.

remove(*val*)

Remove an element. If not a member, raise a KeyError.

update(*args*)

Update a set with the union of itself and others.

WriterFactory = *<pyomo.common.factory.Factory object>*

_default_nl_writer

alias of `NLWriter`

_apply_to(*instance*, ***kwds*) [source]

ccats_common.common_debug.**compute_infeasibility_explanation**(*model*, *solver=None*, *tee=False*, *tolerance=1e-08*) [source]

PYOMO DEBUG FUNCTION FROM PYOMO DEVELOPMENT TEAM

This function attempts to determine why a given model is infeasible. It deploys two main algorithms:

1. Successfully relaxes the constraints of the problem, and reports to the user some sets of constraints and variable bounds, which when relaxed, creates a feasible model.
2. Uses the information collected from (1) to attempt to compute a Minimal Infeasible System (MIS), which is a set of constraints and variable bounds which appear to be in conflict with each other. It is

minimal in the sense that removing any single constraint or variable bound would result in a feasible subsystem.

Parameters:
- **model** (*A pyomo block*)
- **(optional)** (*tolerance*)
- **(optional)**
- **(optional)** – constraint feasible (1e-08)

`ccats_common.common_debug.`**`_get_results_with_value`**`(`*`constr_value_generator, msg=None`*`)` [source]

`ccats_common.common_debug.`**`_get_results`**`(`*`constr_generator, msg=None`*`)` [source]

`ccats_common.common_debug.`**`_get_constraint`**`(`*`modified_model, v`*`)` [source]

ccats_common.common_pytest module

Run Pytest for CCATS

Common Pytest: Summary

CCATS utility submodule for running Pytest.

Common Pytest: Input Files

None

Common Pytest: Model Functions and Class Methods

- `__init__()` - Initialize variables for running Pytest.
- `run()` - Run Pytest.

Common Pytest: Output Debug Files

- ccats_common//tests_files//logs//**pytestlog_<model_year>.log** - log of Pytest results where <model_year> is the year of the current run.

Common Pytest: Output Restart Variables

Common Pytest: Code

class `ccats_common.common_pytest.`**`TestCollection`**`(`*`parent`*`)` [source]

Bases: `object`

Class for running Pytest for CCATS.

`__init__``(`*`parent`*`)` [source]

Initializes TestCollection object.

Parameters: **parent** (*Class*) – Parent class for Pytest model testing.

Return type: None

run() [source]

Run Pytest for CCATS.

Parameters: **None**
Return type: None

ccats_common.common_visual module

Utility file containing common visualization functions used in CCATS.

Common Visual: Summary

This is a utility file containing visualization functions used in CCATS.

Common Visual: Input Files

None

Common Visual: Model Functions and Class Methods

- `create_folium_map()` - Map a CCATS solution.
- `print_datatable_to_html()` - Create an html data table

Common Visual: Output Debug Files

- `create_folium_map()` outputs a map in a user specified path and filename.
- `print_datatable_to_html()` outputs a table in a user specified path and filename.

Common Visual: Output Restart Variables

None

Common Visual: Code

ccats_common.common_visual.**create_folium_map**(*df*, *outdir*, *map_name='pipeline_map.html'*)

Create a folium map. [source]

Parameters:
- **df** (*DataFrame*) – Dataframe with node, pipeline data, and co2 volume.
- **outdir** (*str*) – Path for saving the map.
- **map_name** (*str*) – Filename for the map.

Return type: None

ccats_common.common_visual.**print_datatable_to_html**(*df*, *outfile*, *headear=None*) [source]

Print map input datatable to html format.

Parameters:
- **df** (*DataFrame*) – Data to map.
- **outdir** (*str*) – Path for saving the map.
- **map_name** (*str*) – Filename for the map.

Return type: None

ccats_common.folium_objects module

Utility file containing folium objects used in CCATS for mapping.

Folium Objects: Summary

This is a utility file containing folium objects used in CCATS for mapping.

Folium Objects: Input Files

None

Folium Objects: Model Functions and Class Methods

- `create_marker()` - creates folium markers (called by `ccats_common.folium_pipeline_map.create_nodes()`).
- `create_pipeline()` - creates folium polylines to represent pipelines between two points (called by `ccats_common.folium_pipeline_map.create_pipelines()`).
- `create_arrowline()` - creates folium polyline to represent pipelines between two points (called by `ccats_common.folium_pipeline_map.create_pipelines()`).
- `create_dashedline()` - creates dashed folium polyline to represent pipelines between two points (called by `ccats_common.folium_pipeline_map.create_pipelines()`).
- `create_legend()` - creates legend for folium map (called by `run()`).
- `export_legend()` - saves legend (called by `create_legend()`).

Folium Objects: Output Debug Files

None

Folium Objects: Output Restart Variables

None

Folium Objects: Code

`ccats_common.folium_objects.`**`create_marker`**`(`*`lat, long, node_types, color='red', radius=5.0, popup_data='N/A', featureGroupName=None`*`)` [source]

Creates Folium markers.

Parameters:
- **lat** (*float*) – Latitude
- **long** (*float*) – Longitude
- **node_types** (*list*) – List of folium featuregroups representing node groups
- **color** (*string*) – Color for marker
- **radius** (*float*) – Radius for marker
- **popup_data** (*str*) – Data shown in popup when marker is clicked
- **featureGroupName** (*str*) – Tells with node_types to add marker to

Returns: **Node_types with marker representing each node**

Return type: list

ccats_common.folium_objects.**create_pipeline**(*id*, *volume*, *pointA*, *pointB*, *color*, *weight=1*) [source]

Creates Folium polylines to represent pipelines between two points.

Parameters:
- **id** (*str*) – Pipeline ID.
- **volume** (*float*) – CO_2 volume
- **pointA** (*list*) – Lat and long for a point.
- **pointB** (*list*) – Lat and long for a point.
- **color** (*str*) – Color of the pipeline.
- **weight** (*int*) – Weight of line.

Return type: Folium polyline

ccats_common.folium_objects.**create_arrowline**(*color*, *path*, *weight=24*) [source]

Creates Folium polyline to represent pipelines between two points.

Parameters:
- **color** (*str*) – Color for the line.
- **path** (*Folium Polyline*) – Folium Polyline to render.
- **weight** (*int*) – Weight of line

Return type: Folium PolyLineTextPath

ccats_common.folium_objects.**create_dashedline**(*color*, *path*, *weight=24*) [source]

Creates dashed folium polyline to represent pipelines between two points.

Parameters:
- **color** (*str*) – Color for the line.
- **path** (*Folium Polyline*) – Folium Polyline to render.
- **weight** (*int*) – Weight of line

Return type: Folium PolyLineTextPath

ccats_common.folium_objects.**create_legend**(*color_map*, *fname*) [source]

Creates legend for folium map.

Parameters:
- **color_map** (*dict*) – Color assigned to each node type.
- **fname** (*str*) – Name to save the legend output file.

Return type: None

ccats_common.folium_objects.**export_legend**(*legend*, *filename='legend.png'*) [source]

Saves legend.

Parameters:
- **legend** (*Matplotlib legend*) – Legend to be saved.
- **fname** (*str*) – Name to save the legend output file.

Return type: None

ccats_common.folium_pipeline_map module

Utility file containing Folium functions used in CCATS for mapping.

Folium Pipeline Map: Summary

This is a utility file containing Folium functions used in CCATS for mapping.

Folium Pipeline Map: Model Functions and Class Methods

- `scale_values()` - Creates a width scale for Folium elements based on volume (called by `calculate_node_volumes()` and `calculate_pipeline_volumes()`).
- `create_nodetypes()` - Creates the types of groups to place nodes (called by `run()`).
- `calculate_node_volumes()` - Calculates volume for each node and assign a weight(radius) for Folium object (called by `create_nodes()`).
- `calculate_pipeline_volumes()` - Calculates volume for each pipeline and assign a weight(radius) for Folium object (called by * `create_pipelines()`).
- `create_nodes()` - Iterate through the pipeline DataFrame and create each node (called by `create_nodes_pipelines()`).
- `create_pipelines()` - Iterate through the pipeline DataFrame and create each pipeline (called by `create_nodes_pipelines()`).
- `create_nodes_pipelines()` - Create nodes and pipelines (called by `run()`).
- `run()` - Create a map of CCATS data (called by `ccats_common.common_visual.create_folium_map()`).

Folium Pipeline Map: Input Files

None

Folium Pipeline Map: Output Debug Files

None

Folium Pipeline Map: Output Restart Variables

None

Folium Pipeline Map: Code

`ccats_common.folium_pipeline_map.`**`scale_values`**`(`*`df, column_name, scale_start=1, scale_end=5`*`)` [source]

Creates a width scale for Folium elements based on volume.

Parameters:
- **df** (*DataFrame*) – DataFrame with node, pipeline data, and CO_2 volume.
- **column_name** (*str*) – Name of column to base the scale on, generally will be co2_volume.
- **scale_start** (*int*) – Min value of scale.
- **end** (*scale*) – Max value of scale.

Return type: DataFrame with scale

`ccats_common.folium_pipeline_map.`**`create_nodetypes`**`(`*`types`*`)` [source]

Creates the types of groups to place nodes.

- create a Folium.FeatureGroup for each node type

Parameters: **types** (*None or list*) – Types of nodes to be plotted.

Returns: **List of node groups**

Return type: list

`ccats_common.folium_pipeline_map.`**`calculate_node_volumes`**`(`*`df`*`)` [source]

Calculates volume for each node and assign a weight(radius) for Folium object.

- Create a dataframe with id, volume and weight
- Sum the volume of nodes with the same id
- Assign a weight based on volume for each node id

Parameters: **df** (*DataFrame*) – DataFrame with node, pipeline data, and CO_2 volume

Returns: **DataFrame with id, volume and weight**

Return type: DataFrame

`ccats_common.folium_pipeline_map.`**`calculate_pipeline_volumes`**`(`*`df`*`)` [source]

Calculates volume for each pipeline and assign a weight(radius) for Folium object.

- Create a dataframe with id, volume and weight,
- Sum the volume of j nodes with the same id, and
- Assign a weight based on volume for each node id.

Parameters: **df** (*Dataframe*) – DataFrame with node, pipeline data, and CO_2 volume.

Returns: **DataFrame id, volume and weight**

Return type: DataFrame

`ccats_common.folium_pipeline_map.`**`create_nodes`**`(`*`df, node_types, color_map`*`)` [source]

Iterate through the pipeline dataframe and create each node.

- Calls calculate_node_volumes to get dataframe with weight for each node

  if id is not in dataframe, min value is assigned

- Call create marker to create a Folium marker for each node

Parameters:
- **df** (*DataFrame*) – DataFrame with node, pipeline data, and CO` $_2$ volume.
- **node_types** (*list*) – List of folium.FeatureGroup node groups.
- **color_map** (*dict*) – Color assigned to each node type.

Returns: **node_types with Folium makers representing nodes**

Return type: list

`ccats_common.folium_pipeline_map.`**`create_pipelines`**`(`*`df, color, line_type=None`*`)` [source]

Iterate through the pipeline dataframe and create each pipeline.

- Calls calculate_node_volumes to get dataframe with weight for each pipeline
  - If id is not in dataframe, min value is assigned
- Calls create_pipeline to create a Folium line for each pipeline

Parameters: **df** (*DataFrame*) – DataFrame with node, pipeline data, and CO_2 volume.

Returns: **list of Folium polylines representing pipelines**

Return type: list

`ccats_common.folium_pipeline_map.`**`create_nodes_pipelines`**`(`*`pipeline_data, node_groups, color_map, pipeline_color='orange', line_type=None`*`)` [source]

Create nodes and pipelines.

Parameters:
- **pipeline_data** (*DataFrame*) – Pipeline data for plotting.
- **node_groups** (*list*) – List of node groups
- **color_map** (*dict*) – Dictionary matching node types to their plot colors.
- **pipeline_color** (*str*) – Color for plotting pipelines.
- **line_type** (*None or str*) – Type of lines to be plotted.

Returns:
- **node_groups** (*list*) – List of Folium objects representing nodes.

- **pipelines** (*list*) – List of Folium objects representing pipelines.

`ccats_common.folium_pipeline_map.`**`run`**`(`*`m, df, outdir, outfile`*`)` [source]

Create a map of CCATS results.

Parameters:

- **m** (*Folium Map*) – Map object for plotting data.
- **df** (*list*) – List of DataFrames of data for plotting.
- **outdir** (*str*) – Directory to sainge the map.
- **outfile** (*str*) – Filename for saving the map.

Return type: Folium Map

`ccats_common.folium_pipeline_map.`**`run_preprocessor_map`**`(`*`m, df, outdir, outfile`*`)` [source]

Module contents

ccats_financial module

Class for declaring CCATS Financial Assumptions.

CCATS Financial: Summary

ccats_financial is the preprocessor for model financial assumptions and a utility for Preprocessor: Summary to calculate the present value of model costs by block.

The module operates as follows:

1. Reads in financial assummption inputs in `setup()`.
2. Calculates the CCATS model discount rate in `calculate_discount_rate()`.

`calculate_discount_investment()` and `calculate_discount_variable_policy()` are called from Preprocessor: Summary.

CCATS Financial: Functions and Class Methods¶

- `__init__()` - Constructor to initialize Class (instantiated by `module.Module.setup()` in Module: Summary).
- `setup()` - CCATS_Financial Setup method (called by `module.Module.setup()`).
- `run()` - Runs main CCATS_Financial Processes (called by `module.Module.setup()`).
- `calculate_discount_rate()` - Calculate discount rate for Carbon Capture and Storage Projects (called by `run()`).
- `calculate_discount_investment()` - Compute the present value of an investment (called by `preprocessor.Preprocessor.instantiate_pyomo_series()`).
- `calculate_discount_variable_policy()` - Compute the present value of a variable or policy cost (called by `preprocessor.Preprocessor.instantiate_pyomo_series()`).

CCATS Financial: Input Files

- fin_setup.csv - setup file with model financial inputs (debt rate, block durations, etc.).

CCATS Financial: Output Debug Files

None

CCATS Financial: Output Restart Variables

None

CCATS Financial: Code

class ccats_financial.**CCATS_Finance**(*parent*) [source]

Bases: `object`

Class for declaring CCATS Financial Assumptions.

__init__(*parent*) [source]

Initializes attributes for CCATS_Finance.

Parameters: **parent** (*module.Module*) – Pointer to head module

Return type: None

parent

module.Module head module

setup(*setup_filename*) [source]

Setup function for CCATS financial assumptions.

Parameters: **setup_filename** (*str*) – **ccats_financial** setup file name.

Returns:

- **self.fin_table** (*DataFrame*) – DataFrame of CCATS financial inputs (debt rate, block durations, etc.).
- **self.debt_ratio** (*float*) – Ratio of debt financing.
- **self.return_over_capital_cost** (*float*) – Assumed required return over capital cost for investment.
- **self.fed_tax_rate** (*float*) – Assumed federal tax rate.
- **self.duration_b0** (*int*) – Duration of optimization model block 0.
- **self.duration_b1** (*int*) – Duration of optimization model block 1.
- **self.duration_b2** (*int*) – Duration of optimization model block 2.
- **self.duration_45q** (*int*) – Duration of 45Q tax credit policy for a single facility.
- **self.financing_risk_premia** (*float*) – Assumed risk premia required by investors over risk-free market rate.
- **self.financing_years_transport** (*int*) – Assumed financing years for pipeline transport investments.
- **self.financing_years_storage** (*int*) – Assumed financing years for saline formation storage investments.
- **self.fixed_om_fraction_transport** (*float*) – Fixed O&M costs as a percentage of capital costs for pipeline transport.
- **self.fixed_om_fraction_storage** (*float*) – Fixed O&M costs as a percentage of capital costs for saline formation storage.
- **self.site_dev_years_storage** (*int*) – Assumed number of years for storage site development.

- **self.construction_years_storage** (*int*) – Assumed number of years for storage site construction.
- **self.rate_real** (*NumPy array*) – Array of 10-year treasury rate by year
- **self.rate_inflation** (*NumPy array*) – Array of inflation rate by year.

run() [source]

Run CCATS financial assumptions.

Parameters: **None**

Return type: None

calculate_discount_rate() [source]

Calculate discount rate for Carbon Capture and Storage Projects

Parameters: **None**

Returns:

- **self.expected_inflation_rate** (*NumPy array of floats*) – Inflation rates by year (fraction). If length < n_years, then the last entry is used for missing years.
- **self.rate_discount** (*NumPy array of floats*) – Discount rates by year (fraction). If length < n_years, then the last entry is used for missing years.
- **self.rate_borrowing** (*NumPy array of floats*) – Borrowing rates by year (fraction). If length < n_years, then the last entry is used for missing years.

calculate_discount_investment(*year_invest*, *n_financing*, *fom*) [source]

Compute the present value of an investment.

- Called from Preprocessor: Summary to discount storage and transport investment costs.

Parameters:

- **year_invest** (*int*) – Year that investment decision occurs.
- **n_financing** (*int*) – Number of years that financing is paid over (and duration that Fixed O&M is paid).
- **fom** (*float*) – Fraction of CAPEX paid each year as Fixed O&M.

Returns: **discount** – Relates a lump sum capex investment to its present value (n_years earlier).

Return type: float

calculate_discount_variable_policy(*block_start_year*, *n_years*, *include_inflation*) [source]

Compute the present value of a variable or policy cost.

- Called from Preprocessor: Summary to discount policy, CO_2 EOR net cost offers and opex.

Parameters:

- **year_start** (*int*) – Start year as an index in discount_rates and inflation_rates.

- **n_years** (*int*) – Number of years in the block.
- **include_inflation** (*bool*) – True to include inflation in calculations, False to only include discount rate.

Returns: **discount** – Discounts an annual variable or policy cost payment to its present value taking into account block duration.

Return type: float

ccats_history module

Class for declaring CCATS Historical Data.

CCATS History: Summary

ccats_history is the preprocessor for model historical data, overwriting history with up-to-date inputs.

The module operates as follows:

1. Reads in historical data `setup()`.
2. Sets history of relevant restart variables to zero in `zero_out_history()` then overwrites history with input data in `update_history()`.

CCATS History: Functions and Class Methods

- `__init__()` - Constructor to initialize Class (instantiated by `module.Module.setup()` in Module: Summary).
- `setup()` - CCATS_History Setup method (called by `module.Module.setup()`).
- `run()` - Runs main CCATS_History Processes (called by `module.Module.setup()`).
- `zero_out_history()` - Set history year data of relevant restart variables to 0 (called by `run()`).
- `update_history()` - Overwrite history in relevant restart files with up-to-date inputs (called by `run()`).

CCATS History: Input Files

- hist_setup.csv - setup file with model financial inputs (debt rate, block durations, etc.).
- co2_supply_history_div.csv - CO_2 supply history by census division.
- co2_supply_history_reg.csv - CO_2 supply history by census region.
- co2_eor_history_div.csv - CO_2 EOR demand history by census division.
- co2_eor_history_reg.csv - CO_2 EOR demand history by census region.
- co2_seq_history_div.csv - CO_2 saline formation storage history by census division.
- co2_seq_history_reg.csv - CO_2 saline formation storage history by census region.

CCATS History: Output Debug Files

None

CCATS History: Output Restart Variables

- *CCATSDAT_CO2_sup_out* - CO_2 supply output after optimization, by census division.
- *CCATSDAT_CO2_sup_out_r* - CO_2 supply output after optimization, by census region.
- *CCATSDAT_CO2_seq_out* - CO_2 sequestration output after optimization, by census division.
- *CCATSDAT_CO2_seq_out_r* - CO_2 sequestration output after optimization, by census region.

CCATS History: Code

class ccats_history.**CCATS_History**(*parent*) [source]

Bases: **object**

Class for declaring CCATS Historical Data.

__init__(*parent*) [source]

Initializes CCATS_History object.

Parameters: **parent** (*module.Module*) – Pointer to head module

Return type: None

parent

module.Module head module

setup(*setup_filename*) [source]

Setup function for CCATS financial assumptions.

Parameters: **setup_filename** (*str*) – **ccats_history** setup file name.

Returns:

- **self.co2_supply_hist_div** (*DataFrame*) – DataFrame of CO_2 supply history by census division.
- **self.co2_supply_hist_reg** (*DataFrame*) – DataFrame of CO_2 supply history by census region.
- **self.co2_eor_hist_div** (*DataFrame*) – DataFrame of CO_2 EOR demand history by census division.
- **self.co2_eor_hist_reg** (*DataFrame*) – DataFrame of CO_2 EOR demand history by census region.

- **self.co2_seq_hist_div** (*DataFrame*) – DataFrame of CO_2 saline formation storage history by census division.
- **self.co2_seq_hist_reg** (*DataFrame*) – DataFrame of CO_2 saline formation storage history by census region.

run() [source]

Run CCATS history preprocessor.

Parameters: **None**

Returns:

- **self.co2_supply_hist_div** (*DataFrame*) – DataFrame of CO_2 supply history by census division.
- **self.co2_supply_hist_reg** (*DataFrame*) – DataFrame of CO_2 supply history by census region.

zero_out_history() [source]

Set data for history years equal to 0 in relevant restart variables.

- This is done to ensure that historical dataset only contains volumes from the most up-to-date input files.

Parameters: **None**

Returns:

- **self.parent.rest_co2_sup_out** (*DataFrame*) – Localized copy of restart variable *CCATSDAT_CO2_SUP_OUT*.
- **self.parent.rest_co2_sup_out_r** (*DataFrame*) – Localized copy of restart variable *CCATSDAT_CO2_SUP_OUT_R*.
- **self.parent.rest_co2_seq_out** (*DataFrame*) – Localized copy of restart variable *CCATSDAT_CO2_SEQ_OUT*.
- **self.parent.rest_co2_seq_out_r** (*DataFrame*) – Localized copy of restart variable *CCATSDAT_CO2_SEQ_OUT_R*.

update_history() [source]

Update relevant restart files with historical data from input files.

Parameters: **None**

Returns:

- **self.parent.rest_co2_sup_out** (*DataFrame*) – Localized copy of restart variable *CCATSDAT_CO2_SUP_OUT*.
- **self.parent.rest_co2_sup_out_r** (*DataFrame*) – Localized copy of restart variable *CCATSDAT_CO2_SUP_OUT_R*.
- **self.parent.rest_co2_seq_out** (*DataFrame*) – Localized copy of restart variable *CCATSDAT_CO2_SEQ_OUT*.
- **self.parent.rest_co2_seq_out_r** (*DataFrame*) – Localized copy of restart variable *CCATSDAT_CO2_SEQ_OUT_R*.

ccats_pickle module

Class for handling intermediate variables in CCATS.

CCATS Pickle: Summary

- In year 1 the .pkl files are created, and then in subsequent years they are written to and read.
- The .pkl files are only written out when fcrl = 1 (final regular iteration) or ncrl = 1 (report iteration) to avoid duplication during other NEMS iterations.
- There are therefore two sets of .pkl files, one "fcrl" set for regular iterations, and one "ncrl" set for reporting iterations.
- All *.pkl* filenames and corresponding df names match the relevant corresponding submodule variable names (i.e. preproc_i_storage_xxxx.pkl = self.i_storage_df in Preprocessor: Summary).

The Pickle methods operate as follows:

1. The class is initialized in the `__init__()` method, with class being assigned as a child to Module: Summary here.
2. `read_pkl_vars()` is called from Module: Summary in all model years excluding year 1. In this method all required .pkl local, iterative variable files are read into HSM as class dataframes accessible by all other modules in HSM.
3. `write_pkl_variables()` is called from Module: Summary in all model years. In this method all required local, iterative dataframes are written to .pkl files.

Pickle library documentation: https://docs.python.org/3/library/pickle.html

CCATS Pickle: Functions and Class Methods

- `__init__()` - Constructor to initialize Class (instantiated by `module.Module.setup()` in Module: Summary).
- `read_pkl_vars()` - Read in Pickle local tables (called by `module.Module.setup()`).
- `write_pkl_variables()` - Write out Pickle local tables (called by `output.Output.write_pkl()`).

CCATS Pickle: Input Files

None

CCATS Pickle: Output Debug Files

None

CCATS Pickle: Output Restart Variables

None

CCATS Pickle: Code

class `ccats_pickle.`**`CCATS_Pickle`**`(`*`parent`*`)` [source]

Bases: `object`

Class for handling intermediate variables in CCATS

`__init__``(`*`parent`*`)` [source]

Initializes CCATS_Pickle object.

Parameters: **parent** (*module.Module*) – Pointer to head module

Return type: None

`parent`

module.Module head module

`read_pkl_vars``(`*`itr_type, year_current, temp_filepath`*`)` [source]

Read in Pickle local tables.

Parameters:

- **itr_type** (*str*) – FCRL or NCRL iteration.
- **year_current** (*int*) – Current year being modeled.
- **temp_filepath** (*str*) – Location of .pkl files.

Returns:

- **self.preproc_i_storage_df** (*DataFrame*) – DataFrame of storage formations - input data.
- **self.preproc_i_co2_supply_facility_df** (*DataFrame*) – DataFrame of Co_2 cost curve from NETL - input data.
- **self.preproc_i_pipeline_lookup_df** (*DataFrame*) – DataFrame of Co_2 pipeline lookup table - input data.
- **self.preproc_i_eor_demand_df** (*DataFrame*) – DataFrame of Co_2 EOR site CO2 demanded - input data.
- **self.preproc_i_eor_cost_net_df** (*DataFrame*) – DataFrame of Co_2 EOR net cost for Co_2 - input data.
- **self.preproc_i_ts_multiplier_df** (*DataFrame*) – DataFrame of multipliers for ts-ts node arcs - input data.
- **self.preproc_pipes_existing_df** (*DataFrame*) – DataFrame of existing Co_2 pipeline infrastructure in a given model year.

- **self.preproc_storage_existing_df** (*DataFrame*) – DataFrame of existing Co_2 storage infrastructure in a given model year.
- **self.preproc_co2_facility_eligibility_df** (*DataFrame*) – DataFrame of Co_2 facility 45Q eligibility.
- **self.mod_new_built_pipes_df** (*DataFrame*) – DataFrame of new pipelines built in previous model year Block 1.
- **self.mod_new_aors_df** (*DataFrame*) – DataFrame of new AORS, carried over from the previous model year.
- **self.mod_store_prev_b0_df** (*DataFrame*) – DataFrame of previous model year Co_2 stored in Block 0.

write_pkl_variables(*itr_type*, *year_current*, *temp_filepath*) [source]

Write out Pickle local tables.

Parameters:

- **itr_type** (*str*) – FCRL or NCRL iteration.
- **year_current** (*int*) – Current year being modeled.
- **temp_filepath** (*str*) – Location of .pkl files.

Return type: None

localize_restart module

Submodule for localizing and concatenating restart variables.

Localize Restart: Summary

CCATS receives CO_2 supply and demand from several NEMS modules. Each CO_2 supply or demand source has its own restart file variable. To rationalize and simplify this data, supply and demand data restart variables are re-formatted and concatenated into larger files. The `Localize` submodule accomplishes this as follows:

1. EOR demand and net cost offers are received as DataFrames via the restart file in a base-1 ordered index. In `setup_eor()` these DataFrames are remapped to the HSM play numbers and census division numbers used to identify CO_2 EOR plays.
2. CO_2 supply volumes are aggregated and concatenated into DataFrames *industrial_co2_supply_45q* and *industrial_co2_supply_ntc* in `concat_restart_variables_supply()`
3. CO_2 supply volumes are aggregated and concatenated into DataFrames *industrial_co2_cost_inv* and *industrial_co2_cost_om* in `concat_restart_variables_cost()`.

Localize_restart: Input Files

Localize Restart: Model Functions and Class Methods

- `__init__()` - Constructor to initialize class (instantiated by `module.Module.setup()` in Module: Summary)
- `setup_eor()` - Format EOR demand input files for CCATS preprocessor (called by `module.Module.setup()`).
- `concat_restart_variables_supply()` - Concatenate CO_2 capture type supply restart variables into 45Q and NTC supply tables (called by `module.Module.setup()`).
- `concat_restart_variables_cost()` - Concatenate CO_2 capture type cost restart variables into 45Q and NTC capture cost tables (called by `module.Module.setup()`).

Localize Restart: Output Debug Files

None

Localize Restart: Output Restart Variables

None

Localize Restart: Code

class localize_restart.**Localize**(*parent*) [source]

Bases: `object`

Localize submodule for CCATS.

__init__(*parent*) [source]

Initializes Localize object.

Parameters: **parent** (*str*) – Module.Module (Pointer to parent module)
Return type: None

parent

module.Module head module

setup_eor() [source]

Map EOR demand input files to HSM play numbers and census divisions for CCATS preprocessor.

Returns:
- **self.rest_dem_eor** (*DataFrame*) – DataFrame of CO_2 demand (metric tonnes) from CO_2 EOR.
- **self.rest_cst_eor** (*DataFrame*) – DataFrame of CO_2 price offers (1987$/tonne) from CO_2 EOR.

concat_restart_variables_supply() [source]

Concatenate CO_2 capture type supply restart variables into 45Q and NTC supply tables.

Returns:
- **self.industrial_co2_supply_45q** (*DataFrame*) – DataFrame of industrial CO_2 supply (metric tonnes) eligible for 45Q tax credits.
- **self.industrial_co2_supply_ntc** (*DataFrame*) – DataFrame of industrial CO_2 supply (metric tonnes) not eligible for 45Q tax credits.

concat_restart_variables_cost() [source]

Concatenate CO_2 capture type cost restart variables into 45Q and NTC capture cost tables.

Returns:
- **self.industrial_co2_cost_inv** (*DataFrame*) – Carbon capture investment costs (1987$) by NEMS module and region.
- **self.industrial_co2_cost_om** (*DataFrame*) – Carbon capture O&M costs (1987$) by NEMS module and region.

module module

Parent module of the Carbon Capture, Allocation, Transportation and Sequestration (CCATS) Module.

Module: Summary

The **"Module"** class performs all key model setup functions and runs all sub-class processes (i.e. Restart: Summary, Preprocessor: Summary etc.).

The module operates as follows:

1. Instantiates parent-level variables in `__init__()`
2. Declares parent-level utilities and variables, and runs submodule setup functions in `setup()`, this function operations as follows:

   a. Declare *logger* as class variable,
   b. Declare filepaths,
   c. Read in *setup.csv* and other global input files (i.e. *mapping.csv*),
   d. Run Restart: Summary to get restart variables,
   e. Localize restart variables at the parent class level in `Localize`, and
   f. Run setup functions for the CCATS Preprocessor: Summary, CCATS Optimization: Summary, ref:*Postprocessor* and ref:*Output* modules.

3. Runs CCATS modules

Module: Functions and Class Methods

- `__init__()` - Constructor to initialize Class (called by `ccats.run_ccats()`).
- `setup()` - CCATS Setup method.
- `load_local_parameters()` - Loads in CCATS Parameters from the restart file
- `copy_restart_variables()` - Copies Restart Variables for CCATS from the Restart File into local variables.
- `aggregate_restart_variables()` - Aggregate localized restart variables for Preprocessor: Summary.
- `read_pkl()` - Call CCATS Pickle: Summary to read in pickle files from the previous model year fcrl and ncrl iterations.
- `run()` - Runs main CCATS Model Processes.

Module: Input Files

- **setup.csv** - setup file with key model inputs (model flags, model history year, etc.).
- **mapping.csv** - CCATS regional mapping (mapping states to CCATS regions, etc.).
- **co2_eor_mapping.csv** - Census division and census region mapping for CO_2 EOR plays.
- **idm_mapping.csv** - Proportional allocation of IDM CO_2 capture volumes from census regions to census divisions.
- **co2_supply_index.csv** - Restart file index numbers for CO_2 supply types.
- **co2_seq_index.csv** - Restart file index numbers for CO_2 sequestration types.

Module: Output Debug Files

None

Module: Output Restart Variables

None

Module: Code

class module.**Module** [source]

Bases: `object`

Parent module for CCATS.

__init__() [source]

Initializes Module object.

Parameters: **None**
Return type: None

setup(*main_directory, setup_filename, year, pyfiler1, cycle, scedes*) [source]

CCATS Setup method.

- Imports and declares *pyfiler* object,
- Declares *logger* as parent object executable,
- Assigns input paths and global variables,
- Reads in setup tables,
- Calls Restart: Summary to read in restart variables,
- Calls `load_local_parameters()` to load in model parameters,
- Reads in relevant scedes flags,
- Calls :ref:ccats_pickle: to read in iterative model intermediate variables,

- Calls :ref:ccats_history: to loads in history, and write history data to restart file variables, and
- Calls the Preprocessor: Summary, CCATS Optimization: Summary, Postprocessor: Summary, and Output: Summary *setup* methods.

Parameters:

- **main_directory** (*str*) – Filepath for CCATS main run directory.
- **setup_filename** (*str*) – setup.txt filename.
- **year** (*str or int*) – Current model year.
- **pyfiler1** (*None or pyfiler*) – Tool for reading and writing to the restart file.
- **cycle** (*int*) – Model run cycle from main.py.
- **scedes** (None or scedes file inputs from **main.py**.) – NEMS Scenario Description data.

Returns:

- **self.pyfiler1** (*object*) – Object containing restart file data from **main.py**.
- **self.threads** (*int*) – Number of threads available to run CCATS.
- **self.logger** (*Logger*) – Logger utility.
- ***directories*** (*strings*) – Assorted directory paths including *self.main_directory* and *self.input_path*.
- **self.integrated_switch** (*int*) – Switch indicating whether CCATS run is standalone or integrated.
- **self.setup_table** (*DataFrame*) – Input file containing CCATS model user inputs.
- **self.debug_switch** (*bool*) – Output debug files if True (will result in longer model runtime).
- **self.update_45q_switch** (*bool*) – Update the 45Q policy path with input from setup.txt “eor_45q” and “saline_45q” variables if True.
- **self.visualize_preproc_switch** (*bool*) – Output folium visualizations of optimization inputs if True.
- **self.visualize_postproc_switch** (*bool*) – Output folium visualizations of optimization outputs if True.
- **self.pytest_switch** (*bool*) – Perform pytest internal validation tests if True.
- **self.price_reset_switch** (*bool*) – Reset CO_2 prices to produce endogenous CO_2 prices from models which would otherwise have 0 CO_2 capture if True.
- **self.price_average_switch** (*bool*) – CO_2 prices are set as a weighted average of duals by CD if True.
- **self.price_marginal_switch** (*bool*) – CO_2 prices are set as the marginal CO_2 price by CD if True.
- **self.linear_model_switch** (*bool*) – Model is run as a linear model (vs. a MIP) if True.
- **self.write_restart_before_run_switch** (*bool*) – Output the restart file debug received by CCATS before running if True.
- **self.debug_restart_itr_switch** (*bool*) – Output the restart file debug file every iteration if True (will result in slightly longer model runtime).
- **self.output_restart_unf_switch** (*bool*) – Output the restart file as a .unf after running CCATS if True.

- **self.year_aeo** (*int*) – AEO vintage year.
- **self.year_start** (*int*) – Model start year.
- **self.year_final** (*int*) – Model final year.
- **self.year_base_price** (*int*) – Base dollar year.
- **self.years_steo** (*list*) – Model STEO years.
- **self.eor_45q** (*float*) – EOR 45Q tax credit value.
- **self.saline_45q** (*float*) – Saline Formation storage 45Q tax credit value.
- **self.year_new_45q** (*float*) – Year in which most recent 45Q policy update occurred (2022 Inflation Reduction Act).
- **self.legacy_eor_45q** (*float*) – Legacy 45Q policy EOR tax credit value.
- **self.legacy_saline_45q** (*float*) – Legacy 45Q policy saline formation storage tax credit value.
- **self.year_leg_45q** (*float*) – Year in which legacy 45Q policy update occurred (2018 Bipartisan Budget Act).
- **self.price_reset_value_45q** (*float*) – Default starting price for 45Q eligible CO_2 volumes when model is reset to restart convergence.
- **self.price_reset_value_ntc** (*float*) – Default starting price for 45Q ineligible CO_2 volumes when model is reset to restart convergence.
- **self.supply_select_ts_penalty** (*float*) – Penalty for ts node costs in the model.
- **self.mapping** (*DataFrame*) – DataFrame of NEMS mapping (i.e. HSM regions to LFMM regions).
- **self.co2_eor_mapping** (*DataFrame*) – DataFrame of NEMS mapping to CO_2 EOR projects.
- **self.idm_mapping** (*DataFrame*) – DataFrame of Industrial demand module mapping (Census regions to divisions with distrutions).
- **self.co2_supply_index** (*dict*) – Dictionary of restart file index numbers for CO_2 supply types.
- **self.co2_seq_index** (*dict*) – Dictionary of index numbers for CO_2 sequestration types.

`load_local_parameters()` [source]

Loads in CCATS Parameters from the restart file.

Parameters: None

Returns:

- **self.param_baseyr** (*int*) – Base AEO model year for history (hardcoded as 1990).
- **self.param_fcrl** (*boolean*) – Flag to indicate last regular model iteration.
- **self.param_ncrl** (*boolean*) – Flag to indicate reporting model iteration.

`copy_restart_variables()` [source]

Copy Restart Variables for CCATS from the Restart File into local variables.

- We instantiate local versions of the restart files to ensure data fidelity
- Local variables are instantiated form the restart file exactly once, and written to the restart file exactly once

Parameters: **None**

Returns:

- **self.year_current** (*int*) – Current model year.
- **self.iteration_current** (*int*) – Current model iteration.
- **self.rest_mc_jpgdp** (*DataFrame*) – DataFrame of inflation multipliers.
- **self.rest_mc_rmcorpbaa** (*DataFrame*) – DataFrame of corporate bond rate path.
- **self.rest_mc_rmtcm10y** (*DataFrame*) – DataFrame of ten-year treasury rate path.
- **self.rest_pelin** (*DataFrame*) – DataFrame of industrial electricity prices (1987$/MWh).
- **self.rest_ccs_eor_45q** (*DataFrame*) – DataFrame of 45Q tax credit values for carbon capture from saline formation storage (1987$).
- **self.rest_ccs_saline_45q** (*DataFrame*) – DataFrame of 45Q tax credits values for carbon capture from CO_2 EOR (1987$).
- **self.leg_ccs_eor_45q** (*DataFrame*) – DataFrame of legacy 45Q tax credit values for carbon capture from saline formation storage (1987$).
- **self.rest_i_45q_duration** (*int*) – Tax code section 45Q subsidy duration
- **self.rest_i_45q_syr** (*int*) – Start year of tax code section 45Q subsidy
- **self.rest_i_45q_lyr_ret** (*int*) – End year of tax code section 45Q subsidy for retrofits
- **self.rest_i_45q_lyr_new** (*int*) – End year of tax code section 45Q subsidy for new builds
- **self.leg_ccs_eor_45q** (*df*) – DataFrame of legacy 45Q tax credit values for carbon capture from saline formation storage (1987$)
- **self.leg_ccs_saline_45q** (*DataFrame*) – DataFrame of legacy 45Q tax credits values for carbon capture from CO_2 EOR (1987$).
- **self.eor_45q** (*int*) – Base 45Q tax credit value for carbon capture from CO_2 EOR (1987$).
- **self.saline_45q** (*int*) – Base 45Q tax credit value for carbon capture from saline formation storage (1987$).
- **self.legacy_eor_45q** (*int*) – Legacy base 45Q tax credit value for carbon capture from CO_2 EOR (1987$).
- **self.legacy_saline_45q** (*int*) – Legacy base 45Q tax credit value for carbon capture from saline formation storage (1987$).
- **self.rest_i_45q_duration** (*int*) – 45Q tax credit duration (years).
- **self.rest_i_45q_lyr_ret** (*int*) – Final year for 45Q retrofits.
- **self.rest_i_45q_syr** (*int*) – Start year of 45Q tax credit program.
- **self.rest_dem_eor** (*DataFrame*) – DataFrame of CO_2 demand (metric tonnes) from CO_2 EOR.
- **self.rest_cst_eor** (*DataFrame*) – DataFrame of CO_2 price offers (1987$/tonne) from CO_2 EOR.
- **self.rest_play_map** (*DataFrame*) – DataFrame of CO_2 EOR play number indices.

- **CO:sub:`2` Supply and Capture Cost Variables** (*DataFrames*) – DataFrames of CO_2 supply (metric tonnes) and capture costs (1987$) from other NEMS modules (HSM, HMM, LFMM, IDM, EMM).
- **CO:sub:`2` Price Variables** (*DataFrames*) – CO_2 prices (1987$) calculated in CCATS and sent to other NEMS by Census Region/Division.
- **CCATS Reporting Variables** (*DataFrames*) – CO_2 supply and demand reporting variables.

aggregate_restart_variables() [source]

Aggregate localized restart variables for Preprocessor: Summary.

Parameters: **None**

Returns:

- **self.rest_dem_eor** (*DataFrame*) – DataFrame of CO_2 EOR CO_2 demand (metric tonnes) by play.
- **self.rest_cst_eor** (*DataFrame*) – DataFrame of CO_2 EOR price offers for CO_2 (1987$/tonne) by play.
- **self.rest_industrial_co2_supply_45q** (*DataFrame*) – DataFrame of industrial CO_2 supply (metric tonnes) eligible for 45Q tax credits.
- **self.rest_industrial_co2_supply_ntc** (*DataFrame*) – DataFrame of industrial CO_2 supply (metric tonnes) not eligible for 45Q tax credits.
- **self.rest_industrial_co2_cost_inv** (*DataFrame*) – DataFrame of carbon capture investment costs (1987$) by NEMS module and region.
- **self.rest_industrial_co2_cost_om** (*DataFrame*) – DataFrame of carbon capture O&M costs (1987$) by NEMS module and region.

read_pkl() [source]

Call `ccats_pickle.CCATS_Pickle.read_pkl_vars()` to read in pickle files from the previous model year fcrl and ncrl iterations.

- Different pickle outputs are read in to CCATS depending on whether ncrl=1 (if we are running a reporting loop)
- All non-reporting iterations use the previous model year fcrl=1 iteration variables
- All reporting iterations use the previous model year ncrl=1 iteration variables

Parameters: **None**

Return type: None

run(*year=None*, *iteration=None*) [source]

Runs main CCATS Model Processes.

- Preprocessor: Summary
- CCATS Optimization: Summary
- Postprocessor: Summary
- Output: Summary

Parameters:

- **year** (*int*) – Current model year.
- **iteration** (*int*) – Current model iteration.

Returns:

- **self.year_current** (*DataFrame*) – Current model year.
- **self.iteration_current** (*DataFrame*) – Current model iteration.

opmodels package

Submodules

opmodels.ccats_optimization module

Optimization Submodule.

CCATS Optimization: Summary

This submodule is the main optimization program for CCATS. CCATS treats the problem as either a Linear Program (LP) or Mixed Integer Linear Program (MILP) using Pyomo. The optimization operates as follows:

1. `setup()` is called from Module: Summary.Optimization specific options are set based on inputs from *opt_setup.csv*.
2. `run()` is called from Module: Summary.
3. Data is organized by time period in preparation for creating a Pyomo model.
4. The Pyomo model is created using three blocks to represent three model time periods.
5. Model is solved. If solved as a LP, duals are saved.
6. If model was originally solved as MILP, it is re-solved as a LP (by fixing integer variables) to get duals.

CCATS Optimization: Input Files

- opt_setup.csv - optimization setup file

CCATS Optimization: Model Functions and Class Methods

- `__init__()` - Constructor to initialize Class (called by `module.Module.setup()`)
- `setup()` - Setup function using *opt_setup.csv* (called by `module.Module.setup()`)
- `run()` - runs main model functions in `OptimizationModel` (called by `module.Module.run()`)
- `instantiate_pyomo()` - creates the pyomo model (called by `run()`)
- `prepare_data_for_blocks()` - creates dictionaries of data indexed by block (called by `run()`)
- `instantiate_blocks()` - calls most declare_* functions to set-up the optimization problem, assigning data by block (called by `run()`)
- `declare_sets()` - creates sets (called by `instantiate_pyomo()`)
- `declare_parameters()` - creates parameters (variables held fixed) in Pyomo (called by `instantiate_pyomo()`)

- `declare_variables()` - creates variables in Pyomo (called by `instantiate_pyomo()`)
- `declare_constraints()` - creates constraints in Pyomo (called by `instantiate_pyomo()`)
- `declare_objective()` - creates objective function for a single block (called by `instantiate_pyomo()`)
- `declare_constraints_across_blocks()` - creates constraints across blocks (called by `run()`)
- `declare_objective_across_blocks()` - creates multi-block objective (called by `run()`)
- `solve_optimization()` - solve the optimization problem (called by `run()`)
- `solve_linearized_optimization()` - re-solve the optimization as a linear problem to get duals, if originally solved non-linear (called by `run()`)

CCATS Optimization: Output Debug Files

- **main_opt_log.log** - Optimization log file
- debugmodel_file**ccats_model_20XX.mps** - MPS of 20XX optimization problem
- debugmodel_file**ccats_model_20XX.lp** - LP of 20XX optimization problem
- debugoptimization_inputsdata**data_b#_20XX.pkl** - Pickled data inputs for block # for 20XX

CCATS Optimization: Code

class opmodels.ccats_optimization.**OptimizationModel**(*parent*) [source]

Bases: `Submodule`

Main Optimization Submodule for CCATS.

__init__(*parent*) [source]

Initializes OptimizationModel object.

Parameters: **parent** (*str*) – Module.Module (Pointer to parent module)
Return type: None

setup(*setup_filename*) [source]

Setup Main Optimization Submodule for CCATS.

Parameters: **setup_filename** (*str*) – Path to transport setup file.

Returns:

- **self.opt_check_solver_status** (*bool*) – If True, check the solver status
- **self.troubleshoot_datatypes** (*bool*) – If True, log datatypes to assist with troubleshooting
- **self.debug_opt_log** (*Boolean*) – If True, output solver status to nohup (integrated runs only) and main_opt_log.log (standalone & integrated runs)

run() [source]

Run Main Optimization Submodule for CCATS.

Parameters: **None**

Return type: None

`instantiate_pyomo()` [source]

Run Main Optimization Submodule for CCATS.

Parameters: **None**

Returns: **m** – Instantiated Pyomo model

Return type: Pyomo ConcreteModel

`prepare_data_for_blocks()` [source]

Produce block-level tables for optimization.

Parameters: **None**

Returns:

- **self.m.s_time** (*list*) – List of number of time periods as model blocks
- **self.m.duration** (*list*) – List of duration of each block (years)
- **self.m.co2_supply** (*dict*) – Dictionary of CO_2 supply (metric tonnes) by CO_2 source by block
- **self.m.co2_demand** (*dict*) – Dictionary of CO_2 demand (metric tonnes) by CO_2 demand site by block
- **self.m.co2_demand_cost_net** (*dict*) – Dictionary of CO_2 demand net cost ($1987/tonne tonne) by CO_2 demand site by block
- **self.m.storage_injectivity_existing** (*dict*) – Dictionary of existing CO_2 storage site injectivity (tonnes/year) by block
- **self.m.storage_injectivity_new** (*dict*) – Dictionary of potential new CO_2 storage site injectivity (tonnes/year) by block
- **self.m.storage_net_existing** (*dict*) – Dictionary of existing CO_2 storage site net storage capacity (tonnes) by block
- **self.m.storage_net_adder** (*dict*) – Dictionary of potential new CO_2 storage site net storage capacity (tonnes) by block
- **self.m.storage_aors_available** (*dict*) – Dictionary of remaining storage AORs available by block
- **self.m.capex_transport_base** (*dict*) – Dictionary of transportation CAPEX intercept (base) by block
- **self.m.capex_transport_slope** (*dict*) – Dictionary of transportation CAPEX slope by block
- **self.m.opex_transport_elec** (*dict*) – Dictionary of transportation electricity demand by block
- **self.m.storage_capex** (*dict*) – Dictionary of storage capex (1987$/tonne) by storage site by block
- **self.m.storage_opex** (*dict*) – Dictionary of storage opex (1987$/tonne) by storage site by block
- **self.m.discount_invest_storage** (*dict*) – Dictionary of storage investment discount rates by block
- **self.m.discount_invest_transport** (*dict*) – Dictionary of transportation discount rates by block

- **self.m.discount_variable** (*dict*) – Dictionary of variable discount rates by block
- **self.m.discount_policy** (*dict*) – Dictionary of policy discount rates by block

`instantiate_blocks(mb, t)` [source]

Instantiate a single pyomo block for *m*.

Parameters:
- **mb** (*Pyomo Block*) – Pyomo block to be populated
- **t** (*int*) – Time period

Returns: **self.b** – Block populated with sets, parameters, variables, constraints, and objective function.

Return type: Pyomo Block

`declare_sets(data)` [source]

Declare sets.

Parameters: **data** (*object*) – Data for including within this block.

Returns:
- **self.b.s_nodes_supply** (*Pyomo Set*) – Pyomo set of supplu nodes
- **self.b.s_nodes_trans_ship** (*Pyomo Set*) – Pyomo set of trans-shipment nodes
- **self.b.s_nodes_demand** (*Pyomo Set*) – Pyomo set of demand nodes
- **self.b.s_nodes_sequester** (*Pyomo Set*) – Pyomo set of sequestration nodes (i.e. storage and demand)
- **self.b.s_nodes_storage** (*Pyomo Set*) – Pyomo set of storage nodes
- **self.b.s_nodes** (*Pyomo Set*) – Pyomo set of all nodes
- **self.b.s_arcs** (*Pyomo Set*) – Pyomo set of arcs.
- **self.b.s_arcs_out** (*Pyomo Set*) – Pyomo set of arcs out
- **self.b.s_arcs_in** (*Pyomo Set*) – Pyomo set of arcs in
- **self.b.s_transport_options** (*Pyomo Set*) – Pyomo set of transportation options
- **self.b.s_policy_options** (*Pyomo Set*) – Pyomo set of policy options

`declare_parameters(data)` [source]

Declare parameters.

Parameters: **data** (*object*) – Data for including within this block.

Returns:
- **self.b.p_transport_cap_existing** (*Pyomo Param*) – Parameter for block existing transportation capacities.
- **self.b.p_transport_cap_add_min** (*Pyomo Param*) – Parameter for block transport capacity adder minimum values.
- **self.b.p_transport_cap_add_max** (*Pyomo Param*) – Parameter for block transport capacity adder maximum values.
- **self.b.p_capex_transport_base** (*Pyomo Param*) – Parameter for block transportation capex equation intercept.
- **self.b.p_capex_transport_slope** (*Pyomo Param*) – Parameter for block transportation capex equation slope.

- **self.b.p_opex_transport** (*Pyomo Param*) – Parameter for block transportation variable opex.
- **self.b.p_electricity_demand** (*Pyomo Param*) – Parameter for block transportation electricity demand.
- **self.b.p_capex_storage** (*Pyomo Param*) – Parameter for block storage capex.
- **self.b.p_opex_storage** (*Pyomo Param*) – Parameter for block storage variable opex.
- **self.b.p_co2_demand_cost_net** (*Pyomo Param*) – Parameter for block CO_2 demand net costs.
- **self.b.p_policy_cost** (*Pyomo Param*) – Parameter for block policy costs.
- **self.b.p_co2_supply** (*Pyomo Param*) – Parameter for block CO_2 supply.
- **self.b.p_co2_demand** (*Pyomo Param*) – Parameter for block CO_2 demand.
- **self.b.p_storage_injectivity_existing** (*Pyomo Param*) – Parameter for block existing CO_2 injection/year.
- **self.b.p_storage_injectivity_new** (*Pyomo Param*) – Parameter for block potential new CO_2 injection/year.
- **self.b.p_storage_net_existing** (*Pyomo Param*) – Parameter for block existing storage total capacity.
- **self.b.p_storage_net_adder** (*Pyomo Param*) – Parameter for block potential storage total capacity adder.
- **self.b.p_storage_aors_available** (*Pyomo Param*) – Parameter for block remaining storage injection sites available.
- **self.b.p_duration** (*Pyomo Param*) – Parameter for block duration in years.
- **self.b.p_discount_invest_storage** (*Pyomo Param*) – Parameter for storage investment discount rate.
- **self.b.p_discount_invest_transport** (*Pyomo Param*) – Parameter for transportation investment discount rate.
- **self.b.p_discount_variable** (*Pyomo Param*) – Parameter for variable cost discount rate.
- **self.b.p_discount_policy** (*Pyomo Param*) – Parameter for policy cost discount rate.
- **self.b.transport_buffer** (*Pyomo Param*) – Parameter for transportation build buffer.
- **self.b.storage_buffer** (*Pyomo Param*) – Parameter for storage build buffer.
- **self.b.p_M_supply** (*Pyomo Param*) – Parameter for supply bounds.
- **self.b.p_M_storage** (*Pyomo Param*) – Parameter for storage bounds.
- **self.b.p_M_excess** (*Pyomo Param*) – Parameter for excess (infeasibility) cost.

declare_variables() [source]

Declare variables.

Parameters: **None**

Returns:
- **self.b.v_flow** (*Pyomo Var*) – Pyomo decision variable of block flow through arcs.
- **self.b.v_flow_base** (*Pyomo Var*) – Pyomo decision variable of block flow through existing arcs.
- **self.b.v_flow_add** (*Pyomo Var*) – Pyomo decision variable of block flow through arcs requiring investment.
- **self.b.v_flow_by_policy** (*Pyomo Var*) – Pyomo decision variable of block flow by policy type (i.e. 45Q or no tax credit).
- **self.b.vb_transport_investment** (*Pyomo Var*) – Pyomo binary variable for block new investment.
- **self.b.v_transport_cap_add** (*Pyomo Var*) – Pyomo decision variable of block transport capacity added.
- **self.b.v_storage_investment** (*Pyomo Var*) – Pyomo decision variable of block storage investment added.
- **self.b.v_storage_injectivity** (*Pyomo Var*) – Pyomo decision variable of block storage injectivity.
- **self.b.va_slack_storage** (*Pyomo Var*) – Slack variable for storage capacity.
- **self.b.va_slack_demand_eor** (*Pyomo Var*) – Slack variable for demand capacity.
- **self.b.va_excess_supply** (*Pyomo Var*) – Slack variable for excess (infeasible) supply.
- **self.b.va_excess_ts_in** (*Pyomo Var*) – Slack variable for excess (infeasible) supply flowing into ts nodes.
- **self.b.va_excess_ts_out** (*Pyomo Var*) – Slack variable for excess (infeasible) supply flowing out of ts nodes.

`declare_constraints()` [source]

Declare CCATS block-level constraints.

Parameters: **None**

Returns:
- **c_flow_total_rule** (*Pyomo Constraint*) – Pyomo constraint asserting that total flow == flow from existing pipelines + flow from new pipelines.
- **c_flow_by_policy** (*Pyomo Constraint*) – Pyomo constraint asserting that flow by policy type == total flow.
- **c_flow_balance_supply** (*Pyomo Constraint*) – Pyomo constraint asserting that CO_2 out of CO_2 supply nodes == CO_2 supplied to the model.
- **c_flow_balance_transshipment** (*Pyomo Constraint*) – Pyomo constraint asserting that CO_2 out of trans-shipment nodes == CO_2 into trans-shipment nodes.
- **c_flow_balance_demand_storage** (*Pyomo Constraint*) – Pyomo constraint asserting that CO_2 flow to CO_2 storage doesn't exceed storage

capacity.

- **c_flow_balance_demand_eor** (*Pyomo Constraint*) – Pyomo constraint asserting that CO_2 flow to CO_2 demand doesn't exceed demand.
- **c_t0_flow_base** (*Pyomo Constraint*) – Pyomo constraint asserting that no CO_2 transport capacity can be added in time period 1.
- **c_transport_capacity_added_min** (*Pyomo Constraint*) – Pyomo constraint asserting minimum CO_2 transportation capacity that can be added/arc.
- **c_transport_capacity_added_max** (*Pyomo Constraint*) – Pyomo constraint asserting maximum CO_2 transportation capacity that can be added/arc.
- **c_transport_selection** (*Pyomo Constraint*) – Pyomo constraint asserting only one transportation option allowed to be added/arc.
- **c_vb_transport_investment_dummy** (*Pyomo Constraint*) – Pyomo constraint to constrain vb_transport_investment when solved as a linear model.

declare_objective() [source]

Declare objective function.

Parameters: None

Returns:

- **self.b.e_sum_flow_arcs_in** (*Pyomo Expression*) – Sum of flows into each node.
- **self.b.e_sum_policy** (*Pyomo Expression*) – Sum of policy costs (including tax credits).
- **self.b.e_sum_costs_investment** (*Pyomo Expression*) – Sum of investment costs.
- **self.b.e_sum_costs_variable** (*Pyomo Expression*) – Sum of variable costs.
- **self.b.e_sum_excess** (*Pyomo Expression*) – Costs of slack (infeasibilities).
- **self.b.e_sum_costs** (*Pyomo Expression*) – Costs exclusive of slack (infeasibilities).
- **self.b.e_sum_all_costs** (*Pyomo Expression*) – Costs inclusive of slack (infeasibilities).
- **self.b.objective** (*Pyomo Objective*) – Pyomo block-level objective function minimizing total costs.

declare_constraints_across_blocks() [source]

Declare constraints across blocks.

Parameters: None

Returns:

- **c_flow_added** (*Pyomo Constrtaint*) – Pyomo constraint asserting that block flow added * a transport buffer <= block-level transport capacity added in previous time period

- **c_storage_injectivity** (*Pyomo Constraint*) – Pyomo constraint asserting that stored CO_2 <= new storage capacity + existing storage capacity.
- **c_storage_cumulative_injection** (*Pyomo Constraint*) – Pyomo constraint asserting that storage volumes <= exceed maximum formation capacity.
- **c_storage_aors_available** (*Pyomo Constraint*) – Pyomo constraint asserting that storage investment sites <= remaining storage sites available for investment.

`declare_objective_across_blocks()` [source]

Declare objective across blocks.

Parameters: **None**

Returns:

- **self.m.e_sum_blocks** (*Pyomo Expression*) – Pyomo Expression summing all costs across blocks.
- **self.m.objective_multiblock** (*Pyomo Objective*) – Pyomo objective function minimizing total costs across blocks.

`solve_optimization()` [source]

Solve optimization and commit *self.m* model to parent level variable for use by other CCATS submodules.

Parameters: **None**

Returns:

- **self.m.objective_value** (*Float*) – Copy of multiblock objective result.
- **self.parent.pyomo_model** (*Pyomo model*) – Copy of self.m.
- **self.parent.pyomo_block0** (*Pyomo block*) – Copy of self.m.blocks[0] for easier access in postprocessor.
- **self.parent.pyomo_block1** (*Pyomo block*) – Copy of self.m.blocks[0] for easier access in postprocessor.
- **self.parent.pyomo_block2** (*Pyomo block*) – Copy of self.m.blocks[0] for easier access in postprocessor.
- **self.parent.model_name** (*String*) – Name of optimization model ('ccats_opt').

`solve_linearized_optimization()` [source]

Re-solve the optimization problem after linearizing and fixing investment variables.

Parameters: **None**

Returns:

- **self.parent.pyomo_model** (*Pyomo model*) – Copy of self.m_linear
- **self.parent.pyomo_block0** (*Pyomo block*) – Copy of self.m_linear.blocks[0] for easier access in postprocessor.
- **self.parent.pyomo_block1** (*Pyomo block*) – Copy of self.m_linear.blocks[0] for easier access in postprocessor.
- **self.parent.pyomo_block2** (*Pyomo block*) – Copy of self.m_linear.blocks[0] for easier access in postprocessor.
- **self.parent.model_name** (*String*) – Name of optimization model ('ccats_opt').

Module contents

output module

Submodule for outputting CCATS results.

Output: Summary

This submodule outputs CCATS results:

1. `setup()` and `run()` are called from Module: Summary.
2. Pickle results.
3. Reset prices if "self.parent.price_reset_switch" === True.
4. Prepare variables to be written to restart file.

Output: Input Files

None

Output: Model Functions and Class Methods

- `__init__()` - Initializes variables to be populated by Output (called by `module.Module.setup()`)
- `setup()` - Set-up the submodule (called by `module.Module.setup()`)
- `run()` - Calls remaining Output functions (called by `module.Module.run()`)
- `write_pkl()` - Write results using pickle (called by `run()`)
- `reset_prices()` - Reset prices (called by `run()`)
- `write_local_restart_vars()` - Move local restart variables to global variables (called by `run()`)

Output: Output Debug Files

None

Output: Output Restart Variables

- **tcs45q_ccs_eor_45q** - CO_2 sequestered in EOR.
- **tcs45q_ccs_saline_45q** - CO_2 sequestered in saline storage.
- **tcs45q_i_45q_duration** - Tax code section 45Q subsidy duration
- **tcs45q_i_45q_syr** - Start year of tax code section 45Q subsidy
- **tcs45q_i_45q_lyr_ret** - End year of tax code section 45Q subsidy for retrofits
- **tcs45q_i_45q_lyr_new** - End year of tax code section 45Q subsidy for new builds
- **ccatsdat_co2_elec** - Electricity consumed by CO_2 transport.

- **ccatsdat_co2_prc_dis_45q** - CO_2 for 45Q eligible flow at census district level.
- **ccatsdat_co2_prc_dis_ntc** - CO_2 for 45Q ineligible flow at census district level.
- **ccatsdat_co2_prc_reg_45q** - CO_2 for 45Q eligible flow at census region level.
- **ccatsdat_co2_prc_reg_ntc** - CO_2 for 45Q ineligible flow at census region level.
- **ccatsdat_co2_sup_out** - Total CO_2 supplied to CCATS, by census division.
- **ccatsdat_co2_seq_out** - Total CO_2 sequestered by CCATS, by census division.
- **ccatsdat_co2_sup_out_r** - Total CO_2 supplied to CCATS, by census region.
- **ccatsdat_co2_seq_out_r** - Total CO_2 sequestered by CCATS, by census region.

class output.**Output**(*parent*) [source]

Bases: `Submodule`

Output Submodule for CCATS.

__init__(*parent*) [source]

Initializes Output object.

Parameters: **parent** (*str*) – Module.Module (Pointer to parent module)
Return type: None

parent

module.Module head module

setup() [source]

Setup Output Submodule for CCATS.

Parameters: **None**
Return type: None

run() [source]

Run Output Submodule for CCATS.

Parameters: **None**
Return type: None

write_pkl() [source]

Write local variables to pickle

- Because CCATS is in Python and Main is in Fortran, Python must read/write local variables between model years
- This is done using Pickle
- Different pickle outputs are written depending on whether fcrl = 0, fcrl = 1, or ncrl=1 (is this a reporting loop)
- If fcrl=0, do not write pickle outputs
- If fcrl=1 and ncrl=0, write fcrl pickle outputs
- If ncrl=1, write ncrl pickle outputs

Parameters: **None**

Return type: None

`reset_prices()` [source]

Resets prices

Parameters: **None**

Returns:
- **self.parent.ccatsdat_co2_prc_dis_45q** (*DataFrame*) – CO_2 for 45Q eligible flow at census district level.
- **self.parent.ccatsdat_co2_prc_reg_45q** (*DataFrame*) – CO_2 for 45Q eligible flow at census region level.
- **self.parent.ccatsdat_co2_prc_dis_ntc** (*DataFrame*) – CO_2 for 45Q ineligible flow at census district level.
- **self.parent.ccatsdat_co2_prc_reg_ntc** (*DataFrame*) – CO_2 for 45Q ineligible flow at census region level.

`write_local_restart_vars()` [source]

Write local restart variables to global restart variables.

Parameters: **None**

Returns:
- **self.parent.restart.tcs45q_ccs_eor_45q** (*DataFrame*) – CO_2 sequestered in EOR from legacy common block.
- **self.parent.restart.tcs45q_ccs_saline_45q** (*DataFrame*) – CO_2 sequestered in saline storage from legacy common block.
- **self.parent.restart.tcs45q_i_45q_duration** (*int*) – Tax code section 45Q subsidy duration
- **self.parent.restart.tcs45q_i_45q_syr** (*int*) – Start year of tax code section 45Q subsidy
- **self.parent.restart.tcs45q_i_45q_lyr_ret** (*int*) – End year of tax code section 45Q subsidy for retrofits
- **self.parent.restart.tcs45q_i_45q_lyr_new** (*int*) – End year of tax code section 45Q subsidy for new builds
- **self.parent.restart.ccatsdat_co2_elec** (*DataFrame*) – Electricity consumed by CO_2 transport.
- **self.parent.restart.ccatsdat_co2_prc_dis_45q** (*DataFrame*) – CO_2 for 45Q eligible flow at census district level.
- **self.parent.restart.ccatsdat_co2_prc_dis_ntc** (*DataFrame*) – CO_2 for 45Q ineligible flow at census district level.
- **self.parent.restart.ccatsdat_co2_prc_reg_45q** (*DataFrame*) – CO_2 for 45Q eligible flow at census region level.
- **self.parent.restart.ccatsdat_co2_prc_reg_ntc** (*DataFrame*) – CO_2 for 45Q ineligible flow at census region level.
- **self.parent.restart.ccatsdat_co2_sup_out** (*DataFrame*) – Total CO_2 supplied to CCATS, by census division.

- **self.parent.restart.ccatsdat_co2_seq_out** (*DataFrame*) – Total CO_2 sequestered by CCATS, by census division.
- **self.parent.restart.ccatsdat_co2_sup_out_r** (*DataFrame*) – Total CO_2 supplied to CCATS, by census region.
- **self.parent.restart.ccatsdat_co2_seq_out_r** (*DataFrame*) – Total CO_2 sequestered by CCATS, by census region.

postprocessor module

Submodule for Postprocessing CCATS results.

Postprocessor: Summary

This submodule postprocesses CCATS model results and prepares results to be sent to the restart file:

1. `setup()` is called from Module: Summary, assigning postprocessor price smoothing variables and infeasibility output switches.
2. `run()` is called from Module: Summary.
3. Extract results from Pyomo model, storing results as Pandas DataFrames.
4. Convert dual variables to weighted-average CO2 price variables used by other NEMS modules in `duals_to_df()`.
5. Test for infeasibilities.
6. Write data required for next model year calculations to local variables, and pickle.

6. Prepare results to write to the restart file.
7. Create result summaries.
8. Visualize results.

Postprocessor: Input Files

- postproc_setup.csv - postprocessor setup file

Postprocessor: Model Functions and Class Methods

- `__init__()` - Initializes variables to be populated by Postprocessor (called by `module.Module.setup()`)
- `setup()` - Sets run settings using the setup file (called by `module.Module.setup()`)
- `run()` - Calls remaining Postprocessor functions (called by `module.Module.run()`)
- `variables_to_df()` - Process Pyomo results into Pandas DataFrames (called by `run()`)
- `slacks_to_df()` - Processes slack variables (called by `run()`)
- `assign_flows_to_pipes()` - Aggregates flows by pipeline segment (called by `run()`)
- `binding_constraints_to_df()` - Process binding constraints (called by `run()`)
- `duals_to_df()` - Process dual variables (called by `run()`)
- `electricity_demand_to_df()` - Process electricity consumed (called by `run()`)
- `copy_variables_for_pytest()` - Copy variables for Pytest (called by `run()`)
- `test_flows()` - Test slack variables (called by `run()`)
- `update_existing_pipeline_network()` - Process new pipeline capacity (called by `run()`)
- `update_storage_volumes()` - Process new storage capacity $_2$ stored (called by `run()`)

- `write_restart_file()` - Output variables for the restart file (called by `run()`)
- `write_pkl()` - Output variables for pickling (called by `run()`)
- `results_to_csv()` - Save results to CSV (called by `run()`)
- `summarize_outputs()` - Create high level run summary (called by `run()`)
- `visualize_results()` - Create interactive folium maps and datatable (called by `run()`)

Postprocessor: Output Debug Files

Some debug files are output multiple times during a NEMS run, and use the following convention:

- <model> is the name of the model (default is ccats_opt),
- <block> is the number of the current block (0, 1, or 2),
- <year> is the current model year, for example 2024,
- <iteration> is the current NEMS iteration, and
- <cycle> is the current NEMS cycle.

- In debug// (output by `summarize_outputs()`):
  - **<model>_data_stats_postproc.csv** - High level summary of the run
- In debug//optimization_results//binding_constraints// (output by `run()`):
  - **max_cap_binding_trans_existing_<block>_<year>.csv** - Existing transport capacity constraints that are binding (hit limit)
  - **max_cap_binding_trans_add_<block>_<year>.csv** - Added transport capacity constraints that are binding
  - **max_cap_binding_store_<block>_<year>.csv** - Storage injectivity constraints that are binding
- In debug//optimization_tests// (output by `test_flows()`):
  - **infeasible_supply_b<block>_<year>_<iteration>_<cycle>.csv** - Supply node infeasibilities
  - **infeasible_ts_in_b<block>_<year>_<iteration>_<cycle>.csv** - Infeasibilities into trans-shipment nodes
  - **infeasible_ts_out_b<block>_<year>_<iteration>_<cycle>.csv** - Infeasibilities out of trans-shipment nodes
- In debug//optimization_results// (output by `results_to_csv()`):
  - flows//**b<block>_flows_all_years.csv** - All flows
  - investment_decisions//**<model>_bins_b<block>_<year>.csv** - Pipeline binary investment decisions (only meaningful for MILP)
  - transport_cap_added//**<model>_cap_add_b<block>_<year>.csv** - Transportation capacity added
  - investment_decisions//**<model>_store_aors_b<block>_<year>.csv** - Storage capacity added
  - duals//**<model>_duals_<year>.csv** - Dual variables
- In debug//visualization_results//<model>// (output by `visualize_results()`):

    - **postproc_pipeline_map_<year>_<block>.html** - Map of active CO_2 flows
    - **datatable_postproc_pipeline_map_<year>_<block>.html** - Table summarizing results.
- In debug//electricity_demand// (output by `write_restart_file()`):
    - **electricity_demand.csv** - Overview of electricity consumed for transportation

Postprocessor: Output Restart Variables

- **rest_co2_sup_out** - Supplied CO_2 by census division and type
    - 1 = PP Coal,
    - 2 = PP Natgas,
    - 3 = BECCS,
    - 4 = Natural Gas Processing,
    - 5 = Cement,
    - 6 = Ethanol,
    - 7 = Hydrogen,
    - 8 = Other, and
    - 9 = Total.
- **rest_co2_seq_out** - Sequestered CO_2 by census division and storage type where
    - 1 = EOR, and
    - 2 = Saline.
- **rest_co2_sup_out_r** - Supplied CO_2 by census region and type
    - 1 = PP Coal,
    - 2 = PP Natgas,
    - 3 = BECCS,
    - 4 = Natural Gas Processing,
    - 5 = Cement,
    - 6 = Ethanol,
    - 7 = Hydrogen,
    - 8 = Other, and
    - 9 = Total.
- **rest_co2_seq_out_r** - Sequestered CO_2 by census region and storage type where
    - 1 = EOR, and
    - 2 = Saline.
- **rest_co2_prc_dis_45q** - 45Q CO_2 price by census division
- **rest_co2_prc_dis_ntc** - NTC CO_2 price by census division
- **rest_co2_prc_reg_45q** - 45Q CO_2 price by census region
- **rest_co2_prc_reg_ntc** - NTC CO_2 price by census region
- **rest_co2_elec** - Electricity consumed by pipelines by census region

Postprocessor: Code

class postprocessor.**Postprocessor**(*parent*) [source]

Bases: `Submodule`

Postprocessor Submodule for CCATS.

__init__(*parent*) [source]

Initializes Postprocessor object.

Parameters: **parent** (*str*) – Module.Module (Pointer to parent module)

Return type: None

parent

module.Module head module

setup(*setup_filename*) [source]

Setup postprocessor Submodule for CCATS.

Parameters: **setup_filename** (*str*) – Path to postprocessor setup file.

Returns:

- **self.logger** (*Logger*) – Logger utility, declared in parent.
- **self.setup_table** (*DataFrame*) – Setup table for Preprocessor.py, with input values and switches relating to the submodule
- **self.price_limit** (*float*) – Maximum price (+/-) returned by CCATS in $1987.
- **self.price_peg** (*float*) – Maximum price (+/-) movement allowed per cycle relative to the previous cycle in $1987.
- **self.slack_threshold** (*float*) – Volume threshold at which infeasibilities are recorded in debug outputs
- **self.log_b1_infeasibilities** (*bool*) – Determines whether or not to log infeasibilities for block 1.
- **self.log_b2_infeasibilities** (*bool*) – Determines whether or not to log infeasibilities for block 2.
- **self.infeasibility_threshold_pct** (*float*) – Log infeasibilities if greater than provided percentage of total flow.

run() [source]

Run postprocessor Submodule for CCATS.

Parameters: **None**

Returns:

- **self.m** (*Pyomo Model*) – Pyomo model solved by `models.ccats_optimization.OptimizationModel`.
- **self.b0** (*Pyomo Block*) – Block 0 extracted from self.m.
- **self.b1** (*Pyomo Block*) – Block 1 extracted from self.m.
- **self.b2** (*Pyomo Block*) – Block 2 extracted from self.m.

- **self.O_pipe_bins_b0_df** (*DataFrame*) – Block 0 transportation binary investment decisions.
- **self.O_pipe_bins_b1_df** (*DataFrame*) – Block 1 transportation binary investment decisions.
- **self.O_pipe_bins_b2_df** (*DataFrame*) – Block 2 transportation binary investment decisions.
- **self.O_store_new_aors_b0_df** (*DataFrame*) – Block 0 aors opened.
- **self.O_store_new_aors_b1_df** (*DataFrame*) – Block 1 aors opened.
- **self.O_store_new_aors_b2_df** (*DataFrame*) – Block 2 aors opened.
- **self.O_transport_cap_add_b0_df** (*DataFrame*) – Block 0 transportation capacity added.
- **self.O_transport_cap_add_b1_df** (*DataFrame*) – Block 1 transportation capacity added.
- **self.O_transport_cap_add_b2_df** (*DataFrame*) – Block 2 transportation capacity added.
- **self.O_infeasible_supply_b0_df** (*DataFrame*) – Block 0 infeasible supply.
- **self.O_infeasible_supply_b1_df** (*DataFrame*) – Block 1 infeasible supply.
- **self.O_infeasible_supply_b2_df** (*DataFrame*) – Block 2 infeasible supply.
- **self.O_infeasible_ts_in_b0_df** (*DataFrame*) – Block 0 infeasible flow into trans-shipment nodes.
- **self.O_infeasible_ts_in_b1_df** (*DataFrame*) – Block 1 infeasible flow into trans-shipment nodes.
- **self.O_infeasible_ts_in_b2_df** (*DataFrame*) – Block 2 infeasible flow into trans-shipment nodes.
- **self.O_infeasible_ts_out_b0_df** (*DataFrame*) – Block 0 infeasible flow out of trans-shipment nodes.
- **self.O_infeasible_ts_out_b1_df** (*DataFrame*) – Block 1 infeasible flow out of trans-shipment nodes.
- **self.O_infeasible_ts_out_b2_df** (*DataFrame*) – Block 2 infeasible flow out of trans-shipment nodes.
- **self.O_slack_demand_b0_df** (*DataFrame*) – Block 0 slack demand for EOR nodes.
- **self.O_slack_demand_b1_df** (*DataFrame*) – Block 1 slack demand for EOR nodes.
- **self.O_slack_demand_b2_df** (*DataFrame*) – Block 2 slack demand for EOR nodes.
- **self.O_slack_storage_b0_df** (*DataFrame*) – Block 0 slack storage for saline storage nodes.
- **self.O_slack_storage_b1_df** (*DataFrame*) – Block 1 slack storage for saline storage nodes.
- **self.O_slack_storage_b2_df** (*DataFrame*) – Block 2 slack storage for saline storage nodes.
- **self.O_flows_b0_df** (*DataFrame*) – Block 0 flows assigned to pipelines.
- **self.O_flows_b1_df** (*DataFrame*) – Block 1 flows assigned to pipelines.

- **self.O_flows_b2_df** (*DataFrame*) – Block 2 flows assigned to pipelines.
- **self.C_bind_trans_existing_b0_df** (*DataFrame*) – Block 0 binding existing transportation constraints.
- **self.C_bind_trans_existing_b1_df** (*DataFrame*) – Block 1 binding existing transportation constraints.
- **self.C_bind_trans_existing_b2_df** (*DataFrame*) – Block 2 binding existing transportation constraints.
- **self.C_bind_trans_add_b0_df** (*DataFrame*) – Block 0 binding added transportation constraints.
- **self.C_bind_trans_add_b1_df** (*DataFrame*) – Block 1 binding added transportation constraints.
- **self.C_bind_trans_add_b2_df** (*DataFrame*) – Block 2 binding added transportation constraints.
- **self.C_bind_store_b0_df** (*DataFrame*) – Block 0 binding storage constraints.
- **self.C_bind_store_b1_df** (*DataFrame*) – Block 1 binding storage constraints.
- **self.C_bind_store_b2_df** (*DataFrame*) – Block 2 binding storage constraints.

variables_to_df(*block*, *O_flows_df*, *O_pipe_bins_df*, *O_store_new_aors_df*)

Write pyomo results to DataFrames for use in postprocessors. [source]

Parameters:

- **block** (*Pyomo Block*) – Current Pyomo Block to be analyzed.
- **O_flows_df** (*DataFrame*) – Expected to be an empty DataFrame.
- **O_pipe_bins_df** (*DataFrame*) – Expected to be an empty DataFrame.
- **O_store_new_aors_df** (*DataFrame*) – Expected to be an empty DataFrame.

Returns:

- **O_flows_df** (*DataFrame*) – Flows indexed by arc (node i, node j), 45Q elgibility, and pipeline ID.
- **O_pipe_bins_df** (*DataFrame*) – Transportation binary investment decisions indexed by arc (node i, node j) and pipeline ID.
- **O_store_new_aors_df** (*DataFrame*) – Saline storage added (AORS) indexed by node.
- **O_transport_cap_add_df** (*DataFrame*) – Transportation capacity added indexed by arc (node i, node j) and segment.

slacks_to_df(*block*, *O_infeasible_supply*, *O_infeasible_ts_in*, *O_infeasible_ts_out*, *O_slack_demand*, *O_slack_storage*) [source]

Write pyomo slack variables to DataFrames for use in postprocessors.

Parameters:

- **block** (*Pyomo Block*) – Pyomo Block to be analyzed.
- **O_infeasible_supply** (*DataFrame*) – Empty DataFrame.
- **O_infeasible_ts_in** (*DataFrame*) – Empty DataFrame.
- **O_infeasible_ts_out** (*DataFrame*) – Empty DataFrame.
- **O_slack_demand** (*DataFrame*) – Empty DataFrame.
- **O_slack_storage** (*DataFrame*) – Empty DataFrame.

Returns:

- **O_infeasible_supply** (*DataFrame*) – Infeasible supply by node.
- **O_infeasible_ts_in** (*DataFrame*) – Infeasible flow into a trans-shipment node.
- **O_infeasible_ts_out** (*DataFrame*) – Infeasible flow out of a trans-shipment node.
- **O_slack_demand** (*DataFrame*) – Slack EOR demand by node.
- **O_slack_storage** (*DataFrame*) – Slack saline storage by node.

`assign_flows_to_pipes(`*`O_flows_df, O_pipe_bins_df, time_period`*`)` [source]

Assign flows to the correct pipeline segments.

Parameters:

- **O_flows_df** (*DataFrame*) – Current block flows.
- **O_pipe_bins_df** (*DataFrame*) – Transportation investments.
- **time_period** (*int*) – Block number.

Returns: **O_flows_df** – Current block flows assigned to pipelines.

Return type: DataFrame

`binding_constraints_to_df(`*`block, O_flows_df, C_bind_trans_existing_df, C_bind_trans_add_df, C_bind_store_df, new_aors_df`*`)` [source]

Write Pyomo constraints to DataFrames for use in postprocessors.

Parameters:

- **block** (*Pyomo Block*) – Pyomo block to be analyzed.
- **O_flows_df** (*DataFrame*) – Current block flows.
- **C_bind_trans_existing_df** (*DataFrame*) – Empty DataFrame.
- **C_bind_trans_add_df** (*DataFrame*) – Empty DataFrame.
- **C_bind_store_df** (*DataFrame*) – Empty DataFrame.
- **new_aors_df** (*DataFrame*) – Additional storage capacity (AORS) that has been added.

Returns:

- **C_bind_trans_existing_df** (*DataFrame*) – Binding existing transportation constraints for the current block.
- **C_bind_trans_add_df** (*DataFrame*) – Binding added transportation constraints for the current block.
- **C_bind_store_df** (*DataFrame*) – Binding storage constraints for the current block.

`duals_to_df()` [source]

Translate dual variables to df.

Parameters: **None**

Returns:

- **self.duals_df** (*DataFrame*) – DataFrame of dual variables generated from `models.ccats_optimization.OptimizationModel`.
- **self.shadow_price_df** (*DataFrame*) – DataFrame of all shadow prices indexed by tax credit eligibility (policy_eligibility).
- **self.shadow_price_div_df** (*DataFrame*) – DataFrame of shadow prices at the census division level.
- **self.shadow_price_reg_df** (*DataFrame*) – DataFrame of shadow prices at the census region level.

`electricity_demand_to_df()` [source]

Calculate electricity demand and write to dedicated df by census division.

Parameters: **None**
Returns: **self.electric_demand** – Electricity consumed by census division.
Return type: DataFrame

`test_flows()` [source]

Test if supply slack variables indicate that not all supply was allocated to demand or storage.

If infeasibilities found:

- `postprocessor.Postprocessor.results_to_csv()`
- `postprocessor.Postprocessor.summarize_outputs()`

Parameters: **None**

If infeasibilities found:

- `postprocessor.Postprocessor.results_to_csv()`
- `postprocessor.Postprocessor.summarize_outputs()`

Parameters: **None**
Returns: **O_infeasible_supply** – DataFrame of supply slack variables from optimization.
Return type: DataFrame

`update_existing_pipeline_network()` [source]

Get capacity added for new pipelines and write to DataFrame to be passed between model years.

- Use block 0, because those are the investments selected in the curent model year (first available to use in block 1)

Parameters: **None**
Returns: **self.parent.new_built_pipes_df** – DataFrame of pipeline capacity added.
Return type: DataFrame

`update_storage_volumes()` [source]

Update realized storage volumes passed between model years.

- Use block 0 since it is investment block.

Parameters: **None**
Returns:

- **self.parent.new_aors_df** (*DataFrame*) – Additional storage capacity (AORs) added.
- **self.parent.store_prev_b0_df** (*DataFrame*) – Track amount of CO_2 stored.

`write_restart_file()` [source]

Write results to local restart variable.

- Results are based on block 0

Results use the following indices:

- Supply
  - 1 = PP Coal
  - 2 = PP Natgas
  - 3 = BECCS
  - 4 = Natural Gas Processing
  - 5 = Cement
  - 6 = Ethanol
  - 7 = Hydrogen
  - 8 = Other
  - 9 = Total
- Sequestration
  - 1 = CO_2 EOR
  - 2 = Storage

Parameters: **None**

Returns:

- **self.parent.rest_co2_sup_out** (*DataFrame*) – Supplied CO_2 by census division
- **self.parent.rest_co2_seq_out** (*DataFrame*) – Sequestered CO_2 by census division
- **self.parent.rest_co2_sup_out_r** (*DataFrame*) – Supplied CO_2 by census region
- **self.parent.rest_co2_seq_out_r** (*DataFrame*) – Sequestered CO_2 by census region
- **self.parent.rest_co2_prc_dis_45q** (*DataFrame*) – Price of CO_2 45Q eligible by census district
- **self.parent.rest_co2_prc_dis_ntc** (*DataFrame*) – Price of CO_2 45Q ineligible by census district
- **self.parent.rest_co2_prc_reg_45q** (*DataFrame*) – Price of CO_2 45Q eligible by census region
- **self.parent.rest_co2_prc_reg_ntc** (*DataFrame*) – Price of CO_2 45Q ineligible by census region
- **self.parent.rest_co2_elec** (*DataFrame*) – Electricity consumption by census division

`copy_variables_for_pytest()` [source]

Copy flow variables for Pytest.

- Pytest is used for results testing (i.e. confirming flows in the model don't exceed flows reported via the restart variable)

Parameters: None

Returns:

- **self.parent.pytest.O_flows_b0_df** (*DataFrame*) – Block 0 flows copied for pytest.
- **self.parent.pytest.O_flows_b1_df** (*DataFrame*) – Block 1 flows copied for pytest.
- **self.parent.pytest.O_infeasible_supply_b0_df** (*DataFrame*) – Copy of block 0 infeasible supply df for pytest.
- **self.parent.pytest.O_infeasible_supply_b1_df** (*DataFrame*) – Copy of block 1 infeasible supply df for pytest.
- **self.parent.pytest.O_infeasible_supply_b2_df** (*DataFrame*) – Copy of block 2 infeasible supply df for pytest.
- **self.parent.pytest.transport_add** (*DataFrame*) – Copy of transport capacity add df for pytest.
- **self.parent.pytest.transport_existing** (*DataFrame*) – Copy of transport capacity existing df for pytest.
- **self.parent.pytest.year_start** (*int*) – Copy of start year for pytest.
- **self.parent.pytest.year_current** (*int*) – Copy of current year for pytest.
- **self.parent.pytest.iteration_current** (*int*) – Copy of current iteration for pytest.
- **self.parent.pytest.rest_co2_sup_out** (*DataFrame*) – Supplied CO_2 by census division
- **self.parent.pytest.rest_co2_sup_out_r** (*DataFrame*) – Sequestered CO_2 by census division

`write_pkl()` [source]

Write local variables that need to be passed between model iterations to pickle.

Parameters: None

Returns:

- **self.parent.pkl.mod_new_aors_df** (*DataFrame*) – AORS opened during Block 0.
- **self.parent.pkl.mod_store_prev_b0_df** (*DataFrame*) – Previous block 0 storage.
- **self.parent.pkl.mod_new_built_pipes_df** (*DataFrame*) – New built pipelines.

`visualize_results(`*`O_flows_input, block`*`)` [source]

Visualize optimization results.

Parameters:

- **O_flows_input** (*DataFrame*) – Flows from the current block to be visualized.
- **block** (*String*) – Name of block.

Return type: None

`results_to_csv()` [source]

Write Pyomo results to csv.

Parameters: **None**

Return type: None

summarize_outputs() [source]

Create high level summary of run and output to <model>_data_stats_postproc.csv.

Parameters: **None**

Return type: None

preprocessor module

Submodule for Preprocessing CCATS data during runtime.

Preprocessor: Summary

This submodule preprocesses CCATS data from the restart file and input files and prepares it for use in the main CCATS optimization model. The preprocessor runs as follows:

1. `setup()` is called from Module: Summary.
2. In `setup()`, input variables are read in from *preproc_setupd.csv*, then `load_inputs_restart()`, `load_inputs_csv()` and `load_inputs_pkl()` are called to read in input DataFrames.
3. Other model year 1 setup processes are performed, (i.e. input costs are inflation adjusted in `harmonize_costs_inflation()` and 45Q tax credit values are assigned to demand and storage input DataFrames in `assign_tax_credits()`).
4. Preprocessor: Summary setup ends.
5. `run()` is called from Module: Summary.
6. Instantiate model year process DataFrames:
   a. Main model year CO_2 supply facility pipeline lookup, storage, and EOR demand DataFrames are instantiatied in `instantiate_model_year_dfs()`.
   b. Block-level model year CO_2 supply and EOR demand DataFrames are instantiated in `setup_co2_supply_blocks()` and `setup_co2_eor_blocks()`, respectively.
   c. Model year pipeline infrastructure is declared based on generalized Department of Transportation (DOT) pipeline network and previous model year results in `declare_existing_pipe_infrastructure()`.
7. Produce a point-source CO_2 supply roster for the main CCATS optimization:
   a. CO_2 supply sources from the NETL CCRD database have their prices adjusted based on technology improvement (`apply_tech_rate_capture_costs()`) and transport costs (`add_transport_costs_to_netl_capture_costs()`).
   b. Assign NEMS capture costs to Facility Aggregated Nodes (FANs) in `assign_nems_capture_costs()`.

    c. Match CO_2 supply from NEMS against the point-source facility rosters in *self.ccs_facility_year_df* and *self.cluster_facility_df*, based on facility capture cost, producing block-level CO_2 supply facility rosters for the main optimization.
    d. Set 45Q eligibility status for each block-level CO_2 supply facility roster, and adjust capital costs down for the next model year.

8. Produce a transportation roster for the main CCATS optimization:

    a. Make the dataset sparse in `filter_available_pipelines()` based on selected CO_2 supply faciliites and available CO_2 EOR and saline formation storage sites.
    b. Assign pipeline electricity costs in `assign_electricity_costs()`.

9. Produce a CO_2 saline formation storage roster for the main CCATS optimization

    a. Update saline formation storage capacity and injectivity based on previous model year results

10. Setup inputs to CCATS optimization.

11. Run CCATS preprocessor utiliites (i.e. pickle variables, debug outputs)

Preprocessor: Input Files

- preproc_setup.csv - preprocessor setup file
- storage_formations_lookup.csv - Lookup of saline formation attributes
- co2_facility_lookup.csv - Lookup of CO_2 capture facility attributes
- master_pipeline_lookup_multi.csv - Lookup of CO_2 pipeline transportation attributes
- ts_multiplier.csv - Set of CO_2 multipliers for TS arcs by year
- hsm_eor_centroid.csv - CO_2 demand from HSM centroids during STEO years

Preprocessor: Model Functions and Class Methods

Setup

- `__init__()` - Constructor to initialize Preprocessor submodule (instantiated by `module.Module.setup()` in Module: Summary)
- `setup()` - Setup Preprocessor submodule for CCATS (called by `module.Module.setup()`).
- `load_inputs_restart()` - Load CCATS inputs from Restart File (called by `setup()`).

- `load_inputs_csv()` - Load CCATS inputs from .csv files (called by `setup()`).
- `load_inputs_pkl()` - Load CCATS inputs from .pkl files (called by `setup()`).
- `harmonize_costs_inflation()` - Harmonize model input costs using NEMS inflation multipliers (called by `setup()`).
- `filter_pnw()` - Assess Pacific Northwest (PNW) EMM region's viability for CCS activities (called by `setup()`).

Run

- `run()` - Run Preprocessor Submodule for CCATS (called by `module.Module.run()`).

Model Year Instantiations

- `assign_tax_credits()` - Assign 45Q tax credits values to storage and demand input DataFrames (called by `run()`).
- `instantiate_model_year_dfs()` - Instantiate model year DataFrames (i.e. model year pipeline lookup, available saline formations, etc.) (called by `run()`).
- `account_for_facility_year_operations()` - Account for the number of years a facility has been operational (called by `run()`).
- `setup_co2_supply_blocks()` - Instantiate model DataFrames for CO_2 supply (called by `run()`).
- `setup_co2_eor_blocks()` - Setup model year DataFrames for CO_2 EOR demand (called by `run()`).
- `declare_existing_pipe_infrastructure()` - Declare and update existing pipeline network (called by `run()`).

Produce CO_2 supply source roster

- `apply_tech_rate_capture_costs()` - Apply tech learning rate to NETL capture costs (called by `run()`).
- `assign_nems_capture_costs()` - Assign CO_2 capture cost ($/tonne) for net-new facilities based on restart file variables (called by `run()`).
- `get_co2_facility_rosters()` - Determine rosters of available CO_2 capture facilities (called by `run()`).
- `add_transport_costs_to_netl_capture_costs()` - Prioritize CO_2 capture facilities based on transport connection costs (called by `run()`).
- `determine_point_source_supplies()` - Match CO_2 supplies against the NETL curve and produce a DataFrame of model year supply sources (called by `run()`).
- `determine_facility_eligibility_45q()` - Determine 45Q eligibility for CO_2 capture facilities by model block (called by `run()`).

- `zero_inv_costs_for_retrofit_facilities()` - Zeroes out investment costs for facilities which have been retrofit (called by `run()`).

Update CO_2 transportation roster

- `filter_available_pipelines()` - Remove inactive CO_2 supply, storage and CO_2 EOR nodes from the pipeline network to make more sparse (called by `run()`).
- `assign_electricity_costs()` - Assign electricity costs ($/MWh) for pipelines (called by `run()`).

Update CO_2 saline formation storage roster

- `update_storage_infrastructure()` - Declare and update existing storage infrastructure based on last model year results (called by `run()`).

Setup CCATS Optimization

- `declare_supply_opt()` - Declare supply DataFrames for the main CCATS optimization (called by `run()`).
- `prepare_storage_opt()` - Prepare storage DataFrames for the main CCATS optimization (called by `run()`).
- `prepare_eor_opt()` - Prepare EOR DataFrames for the main CCATS optimization (called by `run()`).
- `setup_optimization()` - Format and prepare data in optimization DataFrames for optimization (called by `run()`).
- `instantiate_pyomo_series()` - Declare Pyomo optimization inputs as series (called by `run()`).

CCATS Preprocessor Utilities

- `write_pkl()` - Write CCATS variables that need to be passed between model iterations via .pkl files (called by `run()`).
- `summarize_inputs()` - Output debug file of preprocessor data summary (called by `run()`).
- `visualize_inputs()` - Visualize preprocessed data (called by `run()`).

Preprocessor: Output Debug Files

Some debug files are output multiple times during a NEMS run, and use the following convention:

- <model> is the name of the model (default is ccats_opt),

  - <block> is the number of the current block (0, 1, or 2),
  - <year> is the current model year, for example 2024,
  - <iteration> is the current NEMS iteration, and
  - <cycle> is the current NEMS cycle.

- In debug// (output by `summarize_inputs()`):
  - **<model>_data_stats_preproc.csv** - High level summary of model inputs

- In debug//tech_learning (output by `apply_tech_rate_capture_costs()`):
  - **tech_learning_capture.csv** - Tech learning rate for carbon capture technologies by year

- In debug//tech_learning (output by `instantiate_pyomo_series()`):
  - **tech_learning_transport_storage.csv** - Tech learning rate for carbon capture technologies by year

- In debug//optimization_inputs//supply (output by `determine_facility_eligibility_45q()`):
  - **before_45q_supply_sources_sparse_<block>_df_<year>.csv** - Point-source CO_2 supply sources before being updated for 45Q eligibility
  - **after_45q_supply_sources_sparse_<block>_df_<year>.csv** - Point-source CO_2 supply sources after being updated for 45Q eligibility

- In debug//optimization_inputs//elec_costs (output by `assign_electricity_costs()`):
  - **elec_prices_<year>_<iteration>_<cycle>.csv** - Electricity prices used by CCATS main optimization

- In debug//optimization_inputs//elec_costs (output by `instantiate_pyomo_series()`):
  - **opex_transport_elec_<year>.csv** - Electricity opex ($/tonne) used by CCATS main optimization
  - **electricity_demand_<year>.csv** - Electricity demand (MWh/tonne) used by ccats main optimization

- In debug//optimization_inputs//storage (output by `instantiate_pyomo_series()`):
  - **storage_existing_opt.csv** - Existing storage capacity in the CCATS main optimization
  - **storage_new_opt.csv** - Potential for new storage capacity in the CCATS main optimization
  - **storage_opt_inputs.csv** - Main storage inputs into the CCATS main optimization storage (storage AORs available and operating, injectivity, etc.)

- In debug//optimization_inputs//transport (output by `instantiate_pyomo_series()`):
  - **pipeline_network_<year>.csv** - Available arcs for pipeline network in the CCATS main optimization

- In debug//financing (output by `instantiate_pyomo_series()`):
  - **discount_multipliers.csv** - Block durations and discount multipliers by variable used in the CCATS main optimization

Preprocessor: Output Restart Variables

None

Preprocessor: Code

class preprocessor.**Preprocessor**(*parent*) [source]

Bases: `Submodule`

Preprocessor Submodule for CCATS.

__init__(*parent*) [source]

Initializes Preprocessor object.

Parameters: **parent** (*str*) – Module.Module (Pointer to parent module).
Return type: None

setup(*setup_filename*) [source]

Setup Preprocessor Submodule for CCATS.

Parameters: **setup_filename** (*str*) – Path to preprocessor setup file.

Returns:

- **self.logger** (*Logger*) – Logger utility, declared in parent.
- **self.input_path** (*str*) – Model input path, declared in parent.
- **self.output_path** (*str*) – Model output path, declared in parent.
- **self.preproc_input_path** (*str*) – Preprocessor submodule input path.
- **self.setup_table** (*DataFrame*) – Setup table for Preprocessor.py, with input values and switches relating to the submodule.
- **self.small_sample_switch** (*DataFrame*) – Model solves for only a single census division for testing if TRUE.
- **self.split_45q_supply_switch** (*DataFrame*) – Split CO_2 supply into 45Q and NTC components if TRUE.
- **self.small_sample_division** (*int*) – Census Division tested when running a "small sample" test.
- **self.block_45q_yr** (*int*) – Decision variable for number of 45Q eligibility years remaining in model (see method: determine_facility_eligibility_45q).
- **self.new_pipes_to_exist_facilities** (*int*) – First year that new pipelines can be built to existing capture facilities (first operational in the following year).
- **self.tech_learn_ammonia** (*float*) – Tech rate for capture at ammonia facilities.
- **self.tech_learn_ethanol** (*float*) – Tech rate for capture at ethanol facilities.

- **self.tech_learn_cement** (*float*) – Tech rate for capture at cement facilities.
- **self.tech_learn_ng_processing** (*float*) – Tech rate for capture at natural gas processing.
- **self.tech_learn_other** (*float*) – Tech rate for capture at all other facilities.
- **self.tech_learn_power** (*float*) – Tech rate for capture at power plants (coal and natural gas).
- **self.tech_learn_transport** (*float*) – Tech rate for carbon transport.
- **self.tech_learn_storage** (*float*) – Tech rate for carbon storage.
- **self.transport_buffer** (*float*) – Buffer (fraction) added above CO_2 flow to transport network during optimization.
- **self.storage_buffer** (*float*) – Buffer (fraction) added above CO_2 flow to storage network during optimization.

`load_inputs_restart()` [source]

Load CCATS inputs from Restart File.

- Read in relevant DataFrames from parent module, and
- Assign instantiate “eligibility_45q” column in each supply DataFrame.

Returns:
- **self.i_industrial_supply_45q_df** (*DataFrame*) – DataFrame of 45Q eligibile CO_2 supply from NEMS.
- **self.i_industrial_supply_ntc_df** (*DataFrame*) – DataFrame of CO_2 supply from NEMS that is not eligibile for a tax credit (NTC = no tax credit).
- **self.i_nems_facility_cost_df** (*DataFrame*) – DataFrame of CO_2 capture costs from NEMS.

`load_inputs_csv()` [source]

Load CCATS inputs from CSV.

- Read in input data from .csvs, and
- Concat EOR demand data from the restart file to EOR centroid data from **hsm_eor_centroid.csv** input file, and filter out any EOR plays that have no demand.

Returns:
- **self.i_storage_df** (*DataFrame*) – DataFrame of storage formations - input data.
- **self.i_co2_supply_facility_df** (*DataFrame*) – DataFrame of CO_2 capture facility attributes from NETL - input data.
- **self.i_pipeline_lookup_df** (*DataFrame*) – DataFrame of CO_2 pipeline lookup - input data.
- **self.i_eor_demand_df** (*DataFrame*) – DataFrame of CO_2 EOR site CO_2 demanded - input data.

- **self.i_eor_cost_net_df** (*DataFrame*) – DataFrame of CO_2 EOR net cost for CO_2 - input data.

`load_inputs_pkl()` [source]

Load CCATS inputs from Pickle.

- CCATS Pickle variables are loaded from CCATS Pickle: Summary after model year 1 using the pickle library (https://docs.python.org/3/library/pickle.html),
- This is done because the CCATS Python environment is not maintained between model iterations, pickling allows us to store working memory between iterations, and
- This code can be deprecated once user objects are implemented in main.py.

Returns:

- **self.i_storage_df** (*DataFrame*) – DataFrame of storage formations - input data.
- **self.i_co2_supply_facility_df** (*DataFrame*) – DataFrame of CO_2 capture facility attributes from NETL - input data.
- **self.i_pipeline_lookup_df** (*DataFrame*) – DataFrame of CO_2 pipeline lookup - input data.
- **self.i_eor_demand_df** (*DataFrame*) – DataFrame of CO_2 EOR site CO_2 demanded - input data.
- **self.i_eor_cost_net_df** (*DataFrame*) – DataFrame of CO_2 EOR net cost for CO_2 - input data.
- **self.pipes_existing_df** (*DataFrame*) – DataFrame of existing CO_2 pipeline infrastructure in a given model year.
- **self.storage_existing_df** (*DataFrame*) – DataFrame of existing CO_2 storage infrastructure in a given model year.
- **self.co2_facility_eligibility_df** (*DataFrame*) – DataFrame of CO_2 facility 45Q eligibility.
- **self.parent.new_built_pipes_df** (*DataFrame*) – DataFrame of new pipelines built in previous model year b1.
- **self.parent.new_aors_df** (*DataFrame*) – DataFrame of new AORS, carried over from the previous model year.
- **self.parent.store_prev_b0_df** (*DataFrame*) – DataFrame of previous model year CO_2 stored in b0.
- **self.parent.co2_supply_prev_b1_df** (*DataFrame*) – DataFrame of previous model year CO_2 supplied in b1.

`harmonize_costs_inflation()` [source]

Harmonize model input costs using NEMS inflation multipliers.

- Inflation calculations are called using the `common.calculate_inflation()` function.

Returns:
- **self.i_nems_facility_cost_df** (*DataFrame*) – DataFrame of industrial facility CO_2 capture cost from NEMS - input data.
- **self.i_pipeline_lookup_df** (*DataFrame*) – DataFrame of CO_2 pipeline lookup - input data.
- **self.i_co2_supply_facility_df** (*DataFrame*) – DataFrame of CO_2 capture facility attributes from NETL - input data.
- **self.i_storage_df** (*DataFrame*) – DataFrame of storage formations - input data.

`filter_pnw()` [source]

Assess Pacific Northwest (PNC) EMM region's viability for CCS activities.

- PNW has lots of electricity generation, but limited CCS options, so we pull FANs (Faciliy Aggregated Nodes) out of the representation if no builds in the previous cycle.

Parameters: **None**

Returns:
- **self.i_co2_supply_facility_df** (*DataFrame*) – DataFrame of CO_2 capture facility attributes from NETL - input data.
- **self.i_pipeline_lookup_df** (*DataFrame*) – DataFrame of CO_2 pipeline lookup - input data.

`assign_tax_credits()` [source]

Assign 45Q tax credits values to storage and demand input DataFrames.

Returns:
- **self.i_storage_df** (*DataFrame*) – DataFrame of storage formations - input data.
- **self.i_eor_demand_df** (*DataFrame*) – DataFrame of EOR demand - input data.

`run()` [source]

Run Preprocessor Submodule for CCATS.

Parameters: **None**

Return type: None

`instantiate_model_year_dfs()` [source]

Instantiate model year DataFrames (i.e. model year pipeline lookup, available saline formations, etc.).

- Pipeline CO_2 supplies and pipeline network change every model year, so need to refresh DataFrames every model year.

Parameters: **None**

Returns:

- **self.pipeline_lookup_df** (*DataFrame*) – DataFrame of model year CO_2 pipeline lookup.
- **self.storage_new_df** (*DataFrame*) – DataFrame of model year potential new storage formations.
- **self.eor_demand_df** (*DataFrame*) – DataFrame of model year CO_2 EOR demand by play.
- **self.eor_cost_net_df** (*DataFrame*) – DataFrame of model CO_2 EOR net cost by play.
- **self.co2_supply_facility_df** (*DataFrame*) – DataFrame of available facilities for carbon capture.

`account_for_facility_year_operations()` [source]

Account for the number of years a facility has been operational.

- Update years_operating and 45Q_eligibility before facilities are selected for the optimization.

Returns: **self.co2_supply_facility_df** – DataFrame of CO_2 facility data and costs - input data.

Return type: DataFrame

`setup_co2_supply_blocks()` [source]

Instantiate block-level model DataFrames for CO_2 supply.

- Instantiate individual block-level DataFrames for CO2 supply by facility type, and
- Instantiate a single DataFrame for model year capture costs, with
  - Block 0 CO_2 supply == model year CO_2 supply,
  - Block 1 CO_2 supply == model year + 1 CO_2 supply, and
  - Block 2 CO_2 supply == mean(model year + 2 max(final AEO model year, model year + 7)) CO_2 demand.

Returns:

- **self.year_industrial_supply_b0_df** (*DataFrame*) – DataFrame of model year CO_2 supply for model time period 0.
- **self.year_industrial_supply_b1_df** (*DataFrame*) – DataFrame of model year CO_2 supply for model time period 1.
- **self.year_industrial_supply_b2_df** (*DataFrame*) – DataFrame of model year CO_2 supply for model time period 2.
- **self.year_nems_facility_cost_df** (*DataFrame*) – DataFrame of model year CO_2 capture costs from NEMS modules.

`setup_co2_eor_blocks()` [source]

Setup model year DataFrames for CO_2 EOR demand.

- Block 0 CO_2 demand == model year CO_2 demand,
- Block 1 CO_2 demand == model year + 1 CO_2 demand, and
- Block 2 CO_2 demand == mean(model year + 2 max(final AEO model year, model year + 7) CO_2 demand.

Returns:
- **self.eor_demand_df** (*DataFrame*) – DataFrame of model year CO_2 EOR demand by play.
- **self.eor_cost_net_df** (*DataFrame*) – DataFrame of model CO_2 EOR net cost by play.

`declare_existing_pipe_infrastructure()` [source]

Declare and update existing pipeline network.

- If first model year, instantiate existing pipeline infrastructure based on generalized DOT network, and
- After first model year, update existing transport infrastructure with previous model year's results (pipelines bult in b1).

Parameters: **None**

Returns: **self.pipes_existing_df** – DataFrame of model year existing CO_2 pipeline infrastructure.

Return type: DataFrame

`apply_tech_rate_capture_costs()` [source]

Apply tech learning rate to NETL capture costs.

Parameters: **None**

Returns: **self.co2_supply_facility_df** – DataFrame of available CO_2 facilities for carbon capture.

Return type: DataFrame

`assign_nems_capture_costs()` [source]

Assign CO_2 capture cost ($/tonne) for net-new facilities based on restart file variables.

Parameters: **None**

Returns: **self.co2_supply_facility_df** – DataFrame of available CO_2 facilities for carbon capture.

Return type: DataFrame

`get_co2_facility_rosters()` [source]

Produce model year rosters of available CO_2 capture facilities.

Parameters: **None**

Returns:
- **self.ccs_facility_year_df** (*DataFrame*) – DataFrame of existing carbon capture facilities eligible for carbon capture retrofit.

- **self.cluster_facility_df** (*DataFrame*) – DataFrame of representative clustered carbon capture facilities.

add_transport_costs_to_netl_capture_costs() [source]

Prioritize CO_2 capture facilities based on transport connection costs.

- Update NETL costs to represent capture cost by facility + transport to the closest site, and
- If the closest site isn't source-to-sink, double cost.

Parameters: **None**

Returns: **self.ccs_facility_year_df** – DataFrame of existing carbon capture facilities eligible for carbon capture retrofit.

Return type: DataFrame

determine_point_source_supplies() [source]

Match CO\` $_2$ supplies against the NETL supply curve and produce block-level DataFrames of model year supply sources.

Parameters: **None**

Returns:

- **self.supply_sources_sparse_b0_df** (*DataFrame*) – DataFrame of model year CO_2 supply sources for time period 0.
- **self.supply_sources_sparse_b1_df** (*DataFrame*) – DataFrame of model year CO_2 supply sources for time period 1.
- **self.supply_sources_sparse_b2_df** (*DataFrame*) – DataFrame of model year CO_2 supply sources for time period 2.

determine_facility_eligibility_45q() [source]

Determine 45Q eligibility for CO_2 capture facilities by model block.

- Calculate remaining years of 45Q eligibility by CO_2 supply source by block:
  - If self.split_45q_supply_switch == True, split each CO_2 facility supply into 45Q and NTC components by year, else determine whether flow is 45Q or NTC based on yr_remain_45q and self.block_45q_yr,
  - Concatenate new CO_2 capture faciliites to *co2_facility_eligibility_df* and update years operating.

Parameters: **None**

Returns:

- **self.supply_sources_sparse_b0_df** (*DataFrame*) – DataFrame of time period 0 sparse CO_2 supply sources for model year optimization.
- **self.supply_sources_sparse_b1_df** (*DataFrame*) – DataFrame of time period 1 sparse CO_2 supply sources for model year optimization.

- **self.supply_sources_sparse_b2_df** (*DataFrame*) – DataFrame of time period 2 sparse CO_2 supply sources for model year optimization.
- **self.co2_facility_eligibility_df** (*DataFrame*) – DataFrame of model year CO_2 capture facility 45Q eligibility.

`zero_inv_costs_for_retrofit_facilities()` [source]

Zero out investment costs for facilities which have been retrofit, and establish facility ranking to maintain facility selection

consistency between model years.

- Set capital costs for CO_2 facilities which have already been retrofit to 0.01% of initial costs to maintain sort order in facility roster.

Parameters: **None**

Returns: **self.i_co2_supply_facility_df** – DataFrame of CO2 capture facility attributes from NETL - input data.

Return type: DataFrame

`filter_available_pipelines()` [source]

Remove inactive CO_2 supply, storage and CO_2 EOR nodes from the pipeline network to make more sparse.

Parameters: **None**

Returns: **pipeline_lookup_df** – DataFrame of model year CO_2 pipeline lookup.

Return type: DataFrame

`assign_electricity_costs()` [source]

Assign electricity costs ($/MWh) for pipelines.

Parameters: **None**

Returns:

- **self.pipes_existing_df** (*DataFrame*) – DataFrame of model year existing CO_2 pipeline infrastructure.
- **self.pipeline_lookup_df** (*DataFrame*) – DataFrame of model year CO_2 pipeline lookup.

`update_storage_infrastructure()` [source]

Declare and update existing storage infrastructure based on last model year results.

- If first model year, instantiate existing storage sites, and
- After first model year, merge prior year model results to existing storage infrastructure and update relevant storage site attributes.

Parameters: **None**

Returns: **self.storage_existing_df** – DataFrame of existing CO_2 storage infrastructure in a given model year.

Return type: DataFrame

`declare_supply_opt()` [source]

Declare supply DataFrames for the main CCATS optimization.

Parameters: **None**

Returns:
- **self.supply_sources_sparse_b0_df** (*DataFrame*) – DataFrame of time period 0 sparse CO_2 supply sources for model year optimization.
- **self.supply_sources_sparse_b1_df** (*DataFrame*) – DataFrame of time period 1 sparse CO_2 supply sources for model year optimization.
- **self.supply_sources_sparse_b2_df** (*DataFrame*) – DataFrame of time period 2 sparse CO_2 supply sources for model year optimization.
- **self.co2_supply_b0_df** (*DataFrame*) – DataFrame of time period 0 CO_2 supply sources and volumes for the main optimization.
- **self.co2_supply_b1_df** (*DataFrame*) – DataFrame of time period 1 CO_2 supply sources and volumes for the main optimization.
- **self.co2_supply_b2_df** (*DataFrame*) – DataFrame of time period 2 CO_2 supply sources and volumes for the main optimization.

`prepare_storage_opt()` [source]

Prepare storage DataFrames for the main CCATS optimization.

Parameters: **None**

Returns:
- **self.storage_existing_df** (*DataFrame*) – DataFrame of existing CO_2 storage infrastructure in a given model year.
- **self.storage_new_df** (*DataFrame*) – DataFrame of potential new CO_2 storage infrastructure.
- **self.storage_costs_df** (*DataFrame*) – DataFrame of CO_2 storage costs.

`prepare_eor_opt()` [source]

Prepare EOR dataframes for the main CCATS optimization.

Parameters: **None**

Returns:
- **self.eor_demand_df** (*DataFrame*) – DataFrame of model CO_2 EOR net cost by play.
- **self.eor_cost_net_df** (*DataFrame*) – DataFrame of model year EOR demand net costs.

`setup_optimization()` [source]

Format and prepare data in optimization DataFrames for optimization.

Parameters: **None**

Returns:
- **self.co2_supply_b0_opt** (*DataFrame*) – Optimization DataFrame of time period 0 CO_2 supply sources and volumes for the main optimization.

- **self.co2_supply_b1_opt** (*DataFrame*) – Optimization DataFrame of time period 1 CO_2 supply sources and volumes for the main optimization.
- **self.co2_supply_b2_opt** (*DataFrame*) – Optimization DataFrame of time period 2 CO_2 supply sources and volumes for the main optimization.
- **self.storage_existing_opt** (*DataFrame*) – Optimization DataFrame of existing CO_2 storage infrastructure in a given model year.
- **self.storage_new_opt** (*DataFrame*) – Optimization DataFrame of potential new CO_2 storage infrastructure.
- **self.storage_costs_opt** (*DataFrame*) – Optimization DataFrame of CO_2 storage costs.
- **self.eor_demand_opt** (*DataFrame*) – Optimization DataFrame of model year EOR demand.
- **self.eor_cost_net_opt** (*DataFrame*) – Optimization DataFrame of model year EOR demand net costs.
- **self.pipeline_lookup_opt** (*DataFrame*) – Optimization DataFrame of pipeline network options for optimization.
- **self.pipes_existing_opt** (*DataFrame*) – Optimization DataFrame of existing pipeline infrastructure for optimization.

`instantiate_pyomo_series()` [source]

Declare pyomo optimization inputs as Series.

Parameters: **None**

Returns:

- **self.nodes_supply** (*Series*) – Supply nodes.
- **self.nodes_trans_ship** (*Series*) – Trans-shipment (TS) nodes.
- **self.nodes_sequester** (*Series*) – Sequestration nodes.
- **self.nodes_demand** (*Series*) – EOR Demand nodes.
- **self.nodes_storage** (*Series*) – Storage nodes.
- **self.nodes** (*Series*) – All optimization nodes.
- **self.arcs** (*Series*) – All optimization arcs.
- **self.policy_cost** (*Series*) – CO_2 policy cost.
- **self.eligibility_45q** (*Series*) – Arc eligibility for 45Q.
- **self.transport_existing** (*Series*) – Transportation capacity that does not require investment decision (metric tonnes).
- **self.transport_add_min** (*Series*) – Transport capacity (lower bound) that requires an investment decision (metric tonnes).
- **self.transport_add** (*Series*) – Transport capacity (upper bound) that requires an investment decision (metric tonnes).
- **self.pipeline_lookup_opt_b0** (*Series*) – Transport pipeline lookup for net-new pipeline builds, block 0.
- **self.pipeline_lookup_opt_b1** (*Series*) – Transport pipeline lookup for net-new pipeline builds, block 1.

- **self.pipeline_lookup_opt_b2** (*Series*) – Transport pipeline lookup for net-new pipeline builds, block 2.
- **self.ts_multiplier_b0** (*Series*) – Multiplier for TS arcs to represent added complexity of a regional/national pipeline network, block 0.
- **self.ts_multiplier_b1** (*Series*) – Multiplier for TS arcs to represent added complexity of a regional/national pipeline network, block 1.
- **self.ts_multiplier_b2** (*Series*) – Multiplier for TS arcs to represent added complexity of a regional/national pipeline network, block 2.
- **self.cost_reduce_fr_transport_b0** (*Series*) – Technology improvement rate for transport costs, block 0.
- **self.cost_reduce_fr_transport_b1** (*Series*) – Technology improvement rate for transport costs, block 1.
- **self.cost_reduce_fr_transport_b2** (*Series*) – Technology improvement rate for transport costs, block 2.
- **self.capex_transport_base_b0** (*Series*) – Base capital cost of transportation investment decision used in capex piecewise (1987$), block 0.
- **self.capex_transport_base_b1** (*Series*) – Base capital cost of transportation investment decision used in capex piecewise (1987$), block 1.
- **self.capex_transport_base_b2** (*Series*) – Base capital cost of transportation investment decision used in capex piecewise (1987$), block 2.
- **self.capex_transport_slope_b0** (*Series*) – Capital cost slope of transportation investment decision used in capex piecewise (1987$), block 0.
- **self.capex_transport_slope_b1** (*Series*) – Capital cost slope of transportation investment decision used in capex piecewise (1987$), block 1.
- **self.capex_transport_slope_b2** (*Series*) – Capital cost slope of transportation investment decision used in capex piecewise (1987$), block 2.
- **self.electricity_demand** (*Series*) – Electricity demand from pumps (MWh).
- **self.opex_transport_elec_b0** (*Series*) – Operating cost for electricity ($/MWh), block 0.
- **self.opex_transport_elec_b1** (*Series*) – Operating cost for electricity ($/MWh), block 1.
- **self.opex_transport_elec_b2** (*Series*) – Operating cost for electricity ($/MWh), block 2.
- **self.cost_reduce_fr_storage_b0** (*Series*) – Technology improvement rate for storage costs, block 0.
- **self.cost_reduce_fr_storage_b1** (*Series*) – Technology improvement rate for storage costs, block 1.
- **self.cost_reduce_fr_storage_b2** (*Series*) – Technology improvement rate for storage costs, block 2.
- **self.storage_capex_b0** (*Series*) – Capital and Fixed O&M costs ($/tonne) for saline formation storage, block 0.
- **self.storage_capex_b1** (*Series*) – Capital and Fixed O&M costs ($/tonne) for saline formation storage, block 1.
- **self.storage_capex_b2** (*Series*) – Capital and Fixed O&M costs ($/tonne) for saline formation storage, block 2.

- **self.storage_varom_b0** (*Series*) – Variable O&M costs ($/tonne) for saline formation storage, block 0.
- **self.storage_varom_b1** (*Series*) – Variable O&M costs ($/tonne) for saline formation storage, block 1.
- **self.storage_varom_b2** (*Series*) – Variable O&M costs ($/tonne) for saline formation storage, block 2.
- **self.co2_supply_b0** (*Series*) – CO_2 supply sources and volumes for the main optimization (metric tonnes), block 0.
- **self.co2_supply_b1** (*Series*) – CO_2 supply sources and volumes for the main optimization (metric tonnes), block 1.
- **self.co2_supply_b2** (*Series*) – CO_2 supply sources and volumes for the main optimization (metric tonnes), block 2.
- **self.co2_demand_b0** (*Series*) – CO_2 demand for the main optimization (metric tonnes), block 0.
- **self.co2_demand_b1** (*Series*) – CO_2 demand for the main optimization (metric tonnes), block 1.
- **self.co2_demand_b2** (*Series*) – CO_2 demand for the main optimization (metric tonnes), block 2.
- **self.co2_demand_cost_b0** (*Series*) – CO_2 demand net cost for the main optimization (1987$/tonne), block 0.
- **self.co2_demand_cost_b1** (*Series*) – CO_2 demand net cost for the main optimization (1987$/tonne), block 1.
- **self.co2_demand_cost_b2** (*Series*) – CO_2 demand net cost for the main optimization (1987$/tonne), block 2.
- **self.co2_injectivity_existing_b0** (*Series*) – Existing injectivity for the main optimization (metric tonnes), block 0.
- **self.co2_injectivity_existing_b1** (*Series*) – Existing injectivity for the main optimization (metric tonnes), block 1.
- **self.co2_injectivity_existing_b2** (*Series*) – Existing injectivity for the main optimization (metric tonnes), block 2.
- **self.co2_injectivity_new_b0** (*Series*) – Potential new injectivity for the main optimization (metric tonnes), block 0.
- **self.co2_injectivity_new_b1** (*Series*) – Potential new injectivity for the main optimization (metric tonnes), block 1.
- **self.co2_injectivity_new_b2** (*Series*) – Potential new injectivity for the main optimization (metric tonnes), block 2.
- **self.co2_max_injectivity_b0** (*Series*) – Max injectivity for the main optimization (metric tonnes), block 0.
- **self.co2_max_injectivity_b1** (*Series*) – Max injectivity for the main optimization (metric tonnes), block 1.
- **self.co2_max_injectivity_b2** (*Series*) – Max injectivity for the main optimization (metric tonnes), block 2.
- **self.co2_store_net_existing_b0** (*Series*) – Existing CO_2 storage capacity (metric tonnes), block 0.

- **self.co2_store_net_existing_b1** (*Series*) – Existing CO_2 storage capacity (metric tonnes), block 1.
- **self.co2_store_net_existing_b2** (*Series*) – Existing CO_2 storage capacity (metric tonnes), block 2.
- **self.co2_store_net_adder_b0** (*Series*) – Potential new CO_2 storage capacity (metric tonnes), block 0.
- **self.co2_store_net_adder_b1** (*Series*) – Potential new CO_2 storage capacity (metric tonnes), block 1.
- **self.co2_store_net_adder_b2** (*Series*) – Potential new CO_2 storage capacity (metric tonnes), block 2.
- **self.storage_aors_available_b0** (*Series*) – Remaining storage areas of review available, block 0.
- **self.storage_aors_available_b1** (*Series*) – Remaining storage areas of review available, block 1.
- **self.storage_aors_available_b2** (*Series*) – Remaining storage areas of review available, block 2.
- **self.duration_b0** (*int*) – Time period 0 duration (years).
- **self.duration_b1** (*int*) – Time period 1 duration (years).
- **self.duration_b2** (*int*) – Time period 2 duration (years).
- **self.duration_45q** (*int*) – Duration of 45Q tax credit eligibility (years).
- **self.discount_invest_storage_b0** (*Series*) – Discount rate for investment in storage, block 0.
- **self.discount_invest_storage_b1** (*Series*) – Discount rate for investment in storage, block 1.
- **self.discount_invest_storage_b2** (*Series*) – Discount rate for investment in storage, block 2.
- **self.discount_invest_transport_b0** (*Series*) – Discount rate for investment in transportation, block 0.
- **self.discount_invest_transport_b1** (*Series*) – Discount rate for investment in transportation, block 1.
- **self.discount_invest_transport_b2** (*Series*) – Discount rate for investment in transportation, block 2.
- **self.discount_variable_b0** (*Series*) – Discount rate for variables, block 0.
- **self.discount_variable_b1** (*Series*) – Discount rate for variables, block 1.
- **self.discount_variable_b2** (*Series*) – Discount rate for variables, block 2.
- **self.discount_policy_b0** (*Series*) – Discount rate for policy, block 0.
- **self.discount_policy_b1** (*Series*) – Discount rate for policy, block 1.
- **self.discount_policy_b2** (*Series*) – Discount rate for policy, block 2.

`write_pkl()` [source]

Write CCATS variables that need to be passed between model iterations to .pkl files.

Parameters: **None**

Returns:

- **self.parent.pkl.preproc_i_storage_df** (*DataFrame*) – DataFrame of storage formations - input data.
- **self.parent.pkl.preproc_i_co2_supply_facility_df** (*DataFrame*) – DataFrame of CO_2 cost curve from NETL - input data.
- **self.parent.pkl.preproc_i_pipeline_lookup_df** (*DataFrame*) – DataFrame of CO_2 pipeline lookup table - input data.
- **self.parent.pkl.preproc_i_eor_demand_df** (*DataFrame*) – DataFrame of CO_2 EOR site CO_2 demanded - input data.
- **self.parent.pkl.preproc_i_eor_cost_net_df** (*DataFrame*) – DataFrame of CO_2 EOR net cost for CO_2 - input data.
- **self.parent.pkl.preproc_i_ts_multiplier_df** (*DataFrame*) – DataFrame of multipliers for ts-ts node arcs - input data.
- **self.parent.pkl.preproc_pipes_existing_df** (*DataFrame*) – DataFrame of existing CO_2 pipeline infrastructure in a given model year.
- **self.parent.pkl.preproc_storage_existing_df** (*DataFrame*) – DataFrame of existing CO_2 storage infrastructure in a given model year.
- **self.parent.pkl.preproc_co2_facility_eligibility_df** (*DataFrame*) – DataFrame of CO_2 facility 45Q eligibility.

`summarize_inputs()` [source]

Output debug file of preprocessor data summary.

Parameters: **None**
Return type: None

`visualize_inputs()` [source]

Visualize preprocessed data.

Parameters: **None**
Return type: None

pyscedes module

PyScedesAll Created on April 4 2023 @author: jmw

PyScedesAll is a Python function for parsing the scedes.all file into a user class that can be passed between Python programs for use in NEMS. This code sets a dictionary of scedes keys in the user class.

class `pyscedes.`**`User`**`(`*`scedes_dict`*`)` [source]

Bases: **`object`**

`__init__``(`*`scedes_dict`*`)` [source]

`pyscedes.`**`find_keys_sed`**`()` [source]

`pyscedes.`**`parse_scedes_file`**`(`*`filename`*`)` [source]

Parameters: **NEMS** (*filename- currently hardcoded to the scedes.all file inside of the local folder of*)

Return type: scedes_dict- dictionary of scedes keys

restart module

Class for handling the Restart File in CCATS.

Restart: Summary

The Restart: Summary module reads in restart variables from the restart file, stores these variables in a class dictionary, and writes these CCATS output variables back to the restart file after CCATS processes have been run. The restart file is read from, and written to, using **self.parent.pyfiler1.pyd**, which is maintained by the Integration team. The module operates as follows:

1. `run()` is called from the Module: Summary parent class to read in the restart file.
2. `run()` calls the *read_filer* method from **pyfiler1**. This method loads all the NEMS restart variables into a class dictionary. Using the “output” argument of *read_filer* a second “output” dictionary is also instantiated, indicating which variables need to be output back to the restart file at the end of each CCATS iteration.
3. CCATS restart variables are instantiated, read into appropriately indexed dataframes, and offset by calendar years in the relevant indices. Once this is done, CCATS operations move back to Module: Summary.

5. Restart variables are updated in the various CCATS modules.
6. In the `ccats.run_ccats()` function of ccats the `write_results()` function in Restart: Summary is called.
7. Each instantiated output variable is called from the Restart: Summary class dictionary, re-sized to match restart file formatting, and then written to the appropriate restart variable in the restart file using the *write_filer* method of **pyfiler1**.

Restart: Model Functions and Class Methods

- `__init__()` - Constructor to initialize Restart submodule (instantiated by `module.Module.setup()` in Module: Summary).
- `run()` - Calls the read_filer method from self.parent.pyfiler1 and loads the restart file into CCATS (called by `module.Module.setup()`).
- `add_int()` - Function for adding an integer from the restart file into CCATS as an integer (called by `__init__()`).
- `add_df()` - Function for adding a dataframe from the restart file into NEMS (called by `__init__()`).
- `write_results()` - Repacks local restart file variables into multi-dimensional arrays, and writes to restart file (called by `ccats.run_ccats()`).
- `dump_restart()` - Dumps restart file to .xlsx debug file (called by `write_results()`).

Restart: Output Debug Files

rest_all_<cycle>.xlsx - Debug of CCATS output restart variables, where <cycle> is the current NEMS cycle.

Restart: Code

class `restart.`**`Restart`**`(`*`parent`*`)` [source]

Bases: `object`

Class for handling the Restart File in CCATS

`__init__``(`*`parent`*`)` [source]

Initializes Restart object.

Parameters: **parent** (*str*) – Module.Module (Pointer to parent module)

Return type: None

`parent`

module.Module head module

`output_path`

output path

`df_dict_in`

`df_dict_out`

`int_dict_in`

`int_dict_out`

`run``(`*`temp_filename, temp_rest`*`)` [source]

Calls the read_filer method from **pyfiler1** and loads the restart file into CCATS.

- Determines if CCATS is running integrated in NEMS or standalone.
  - If integrated, pass, since pyfiler1 is imported from **main.py** containing the complete restart file.
  - If standalone, read in the restart file.
- Then instantiate all the restart variables and fill the Restart: Summary class dictionary with the CCATS restart variable keys.

Parameters: **temp_filename** (*str*) – Restart filename.

Returns: **self.__dict__** – Class dictionary of CCATS restart variables read in from the restart file.

Return type: dictionary

`add_int(dict_name, restart_ref, output)` [source]

Function for reading an integer from the restart file into CCATS as an integer.

Parameters:
- **dict_name** (*str*) – Name of restart variable or parameter in **pyfiler1**.
- **restart_ref** (*int*) – Integer data value from **pyfiler1**.
- **output** (*bool*) – Boolean of whether the restart variable is an output value of CCATS.

Returns: **restart_ref** – Integer value from the restart file.

Return type: int

`add_df(dict_name, restart_ref, output, offset=None)` [source]

Function for reading an array from the restart file into CCATS as a DataFrame.

Parameters:
- **dict_name** (*str*) – Name of restart variable or parameter in **pyfiler1**.
- **restart_ref** (*int*) – Integer data value from **pyfiler1**.
- **output** (*bool*) – Boolean of whether the restart variable is an output value of CCATS.
- **offset** (*list*) – List indicating output DataFrame index value offset.

Returns: **df** – Restart variable DataFrame.

Return type: DataFrame

`write_results(temp_filename)` [source]

Repacks local restart file variables into multi-dimensional arrays, and writes these to the restart file.

- Loop through the class dictionary based on keys in *int_dict_out* and *df_dict_out*,
- Split out each of the dictionary keys into different variables,
- Use “getattr” to produce the output array index,
- Set the output array index with “setattr”, and
- These arrays are returned to **main.py** via **pyfiler1**.

Parameters: **temp_filename** (*str*) – Restart File name.

Return type: None

`dump_restart()` [source]

Dumps restart file to .xlsx debug file.

Return type: None

submodule_ccats module

Generic submodule class of CCATS.

- Used to declare variables at the child-level that are universally used across CCATS submodules.
- i.e. self.input_path = self.parent.input_path

Example

import submodule as sub

Submodule: Code

class submodule_ccats.**Submodule**(*parent*, *submodule_name*) [source]

Bases: **object**

Generic submodule for CCATS.

Parameters:
- **parent** (*module.Module*) – Pointer to head module
- **submodule_name** (*str*) – Name of submodule (e.g. ‘Offshore’)

__init__(*parent*, *submodule_name*) [source]

setup(*setup_filename*) [source]

Setup General Submodule for CCATS

Parameters: **setup_filename** (*str*) – Path to submodule setup file.

Examples

def setup(self):

super().setup(setup_filename)

Return type: None

run() [source]

Run General Submodule for CCATS.

Examples

def run(self):

logger.info(‘Running Preprocessor’) super().run()

Return type: None