

---

# **usepa-cti-bca**

***Release 1.0.0***

**OTAQ ASD**

**Feb 07, 2022**

## CONTENTS:

<b>1</b>	<b>Introduction</b>	<b>2</b>
1.1	EPA Heavy-duty Benefit-Cost Analysis (BCA) calculation tool . . . . .	2
1.1.1	What is the BCA tool? . . . . .	2
1.1.2	What are the input files? . . . . .	2
1.1.3	Runtime settings set within the <code>BCA_General_Inputs.csv</code> input file . . . . .	3
1.1.4	What are the output files? . . . . .	4
<b>2</b>	<b>Getting started</b>	<b>5</b>
2.1	Setting up a Python environment . . . . .	5
2.2	Running the tool . . . . .	5
2.3	For help or questions, contact . . . . .	6
<b>3</b>	<b>Methodology</b>	<b>7</b>
3.1	General . . . . .	7
3.2	Calculations and Equations . . . . .	8
3.2.1	Learning effects applied to costs . . . . .	8
3.2.2	Emission repair costs . . . . .	9
3.2.2.1	Direct cost scalers . . . . .	9
3.2.2.2	Estimated warranty & useful life ages . . . . .	10
3.2.2.3	Cost per mile by age (for emission-related repairs) . . . . .	10
3.2.3	Discounting . . . . .	12
3.2.3.1	Present value . . . . .	12
3.2.3.2	Annualized value . . . . .	12
3.3	Sensitivites . . . . .	12
<b>4</b>	<b>usepa-cti-bca bca_tool_code</b>	<b>13</b>
4.1	<code>calc_deltas</code> module . . . . .	13
4.2	<code>def_costs</code> module . . . . .	13
4.3	<code>direct_costs</code> module . . . . .	14
4.4	<code>discounting</code> module . . . . .	16
4.5	<code>emission_costs</code> module . . . . .	16
4.6	<code>figures</code> module . . . . .	17
4.7	<code>fleet_totals_dict</code> module . . . . .	17
4.8	<code>fleet_averages_dict</code> module . . . . .	19
4.9	<code>fuel_costs</code> module . . . . .	20
4.10	<code>general_functions</code> module . . . . .	21
4.11	<code>get_context_data</code> module . . . . .	23
4.12	<code>indirect_costs</code> module . . . . .	24
4.13	<code>project_dicts</code> module . . . . .	25
4.14	<code>project_fleet</code> module . . . . .	25

4.15	repair_costs module . . . . .	26
4.16	sum_by_vehicle module . . . . .	27
4.17	tech_costs module . . . . .	28
4.18	tool_main module . . . . .	28
4.19	vehicle module . . . . .	28
4.20	weighted_results module . . . . .	29
<b>5</b>	<b>Distribution and Support Policy for EPA Software</b>	<b>30</b>
5.1	Copyright Status . . . . .	30
5.2	Disclaimer of Liability . . . . .	30
5.3	Disclaimer of Software Installation/Application . . . . .	30
<b>6</b>	<b>Indices and tables</b>	<b>31</b>
	<b>Python Module Index</b>	<b>32</b>
	<b>Index</b>	<b>33</b>



## INTRODUCTION

### 1.1 EPA Heavy-duty Benefit-Cost Analysis (BCA) calculation tool

#### 1.1.1 What is the BCA tool?

The heavy-duty BCA tool was developed by EPA to estimate costs and benefits (only those based on a dollar-per-ton accounting) of proposed rulemaking options. The tool is written in Python (tested in version 3.10) and makes use of several input files that specify, for example, costs for technology expected to be added to vehicles to facilitate compliance, vehicle populations and sales, fuel consumption, vehicle miles traveled, etc.

#### 1.1.2 What are the input files?

**The list of necessary input files contained in the “inputs” folder is:**

- `BCA_General_Inputs.csv` which specifies which AEO fuel prices to use, what calendar year to which to discount costs, among other parameters.
- `Input_Files.csv` which specifies the specific filenames from which to read-in the necessary runtime data. A user can use different input files/filenames provided they are in the same format as the default files.
- `options_cap.csv` which specifies the criteria air pollutant (CAP) options to be run along with an Option Name for each optionID. Note that “option” and “alternative” and “scenario” tend to be used interchangeably.
- `options_ghg.csv` which specifies the greenhouse gas (GHG) options to be run along with an Option Name for each optionID. Note that “option” and “alternative” and “scenario” tend to be used interchangeably.
- A MOVES-based or fleet file which provides inventories and VMT, etc., to support the CAP analysis.
- A MOVES-based or fleet file which provides inventories and VMT, etc., to support the GHG analysis.
- `MOVES_Adjustments_CAP.csv` which provides adjustments to data in the MOVES-based CAP data file that might be necessary within the BCA tool. Currently, this adjusts regclass 41 diesel data to reflect engine-certs only.
- `MOVES_Adjustments_GHG.csv` which provides adjustments to data in the MOVES-based GHG data file that might be necessary within the BCA tool. Currently, this adjusts applicable sourcetype VPOP data.
- `DirectCostInputs_byRegClass_byFuelType.csv` which provides the direct technology costs by Regulatory Class (used in the CAP analysis).
- `TechCostInputs_bySourceType_byFuelType.csv` which provides the tech costs (direct plus indirect) by SourceType (used in the GHG analysis).

- LearningRateScalars\_byRegClass.csv which provides scalars to be applied in estimating learning effects on direct costs. Currently, this is used in the CAP analysis.
- LearningRateScalars\_bySourceType.csv which provides scalars to be applied in estimating learning effects on direct costs. Currently, this is used in the GHG analysis.
- IndirectCostInputs\_RegClass.csv which provides indirect cost markup factors applied to reg class direct costs to estimate indirect costs; this is used in the CAP analysis.
- IndirectCostInputs\_SourceType.csv which provides indirect cost markup factors applied to sourcetype direct costs to estimate indirect costs. Currently, this is not used.
- ORVR\_FuelChangeInputs\_CAP.csv which provides the CAP fuel consumption impacts expected from adding onboard refueling vapor recovery systems to HD gasoline vehicles.
- ORVR\_FuelChangeInputs\_GHG.csv which provides the GHG fuel consumption impacts expected from adding onboard refueling vapor recovery systems to HD gasoline vehicles.
- DEF\_DoseRateInputs.csv which provides the CAP diesel exhaust fluid (DEF) dosing rates expected in the baseline scenario.
- DEF\_Prices.csv which provides DEF prices by calendar year.
- CriteriaCostFactors.csv which provides the cost per ton of criteria emissions in the inventory (not used for the NPRM analysis).
- Repair\_and\_Maintenance\_Curve\_Inputs.csv which provides inputs used in estimating CAP emission repair costs.
- UsefulLife\_Inputs.csv which provides useful life miles and ages under each CAP option.
- Warranty\_Inputs.csv which provides warranty miles and ages under each CAP option.
- UnitConversions.csv which provides conversion factors as needed by the tool.
- Components\_of\_Selected\_Petroleum\_Product\_Prices.csv which provides fuel prices.
- Table\_1.1.9\_ImplicitPriceDeflators.csv which provides price deflators used by the tool to convert all monetary values to a consistent basis.

### 1.1.3 Runtime settings set within the BCA\_General\_Inputs.csv input file

The user can specify what to run and what AEO fuel prices to use. Runtime settings consist of:

- calculate\_cap\_costs where 'cap' refers to Criteria Air Pollutant and can be set to 'Y' or 'N'.
- calculate\_cap\_pollution\_effects which can be set to 'Y' or 'N' (the default is 'N').
- calculate\_ghg\_costs where 'ghg' refers to Greenhouse Gas and can be set to 'Y' or 'N'.
- calculate\_ghg\_pollution\_effects which can be set to 'Y' or 'N' (this should not be set to 'Y' since necessary inputs are not included).
- dollar\_basis which specifies the dollar basis to use throughout the analysis (input values will be converted to this basis).
- no\_action\_alt which specifies the 'No action' alternative against which any delta calculation will be made (the default is 0).
- aeo\_fuel\_price\_case which specifies the AEO fuel price case to use and can be set to 'Reference', 'High oil price' or 'Low oil price' (the default is 'Reference').

### 1.1.4 What are the output files?

The output files are pretty self-explanatory by their file names.

**Output files generated if calculating CAP costs are:**

- CAP\_bca\_tool\_fleet\_averages.csv which contains average results for all vehicles by calendar year/model year/age.
- CAP\_bca\_tool\_fleet\_totals.csv which contains total results for all vehicles by calendar year/model year/age.
- CAP\_bca\_tool\_annual\_summary.csv which contains annual sums, present values and annualized values using the fleet totals output file.
- CAP\_bca\_tool\_estimated\_ages.csv which contains the required, calculated and estimated warranty and useful life ages.
- CAP\_bca\_tool\_repair\_cpm\_details.csv which contains details of calculations used to estimate repair costs per mile.
- CAP\_bca\_tool\_vmt\_weighted\_emission\_repair\_cpm.csv which contains weighted cost per mile emission repair results by sourcetype/regclass/fueltype.
- CAP\_bca\_tool\_vmt\_weighted\_fuel\_cpm.csv which contains weighted cost per mile fuel costs results by sourcetype/regclass/fueltype.
- CAP\_bca\_tool\_vmt\_weighted\_def\_cpm.csv which contains weighted cost per mile diesel exhaust fluid costs results by sourcetype/regclass/fueltype.

**Output files generated if calculating GHG costs are:**

- GHG\_bca\_tool\_fleet\_averages.csv which contains average results for all vehicles by calendar year/model year/age.
- GHG\_bca\_tool\_fleet\_totals.csv which contains total results for all vehicles by calendar year/model year/age.
- GHG\_bca\_tool\_annual\_summary.csv which contains annual sums, present values and annualized values using the fleet totals output file.

A summary\_log.csv is also created which contains the version number of the tool, date and time statistics for the run and input file data specific to the run.

A folder called “run\_results” will be created within the specific run’s output folder that contains the output files described above. A subfolder called “figures” will be created where figures are saved. A folder called “modified\_inputs” is also created which holds modified versions of the input files. Those modifications include reshaping of the input files along with conversions of the dollar-based inputs into a consistent dollar basis. A folder called “run\_inputs” is also created which holds a direct copy/paste of all input files used for the given run (those specified in Input\_Files.csv). A folder called “code” is also created which holds a direct copy/paste of all files in the bca\_tool\_code package folder (i.e., the python code).

Note that outputs are saved to an outputs folder that will be created (if it does not already exist) in the parent directory of the directory in which the code resides.

## GETTING STARTED

Required files for using the tool can be found in the docket associated with the proposed rulemaking.

In addition, the code repository can be found at [https://github.com/USEPA/EPA\\_HD\\_CTI\\_BCA](https://github.com/USEPA/EPA_HD_CTI_BCA). Navigate to the above link and select “Clone or download” to either clone the repository to your local machine or download it as a ZIP file. Alternatively, you may wish to Fork the repo to your local machine if you intend to suggest changes to the code via a pull request.

### 2.1 Setting up a Python environment

Once you have the tool installed locally and have set up your python environment, you can install the packages needed to use the tool by typing the command (in a terminal window within your python environment):

```
pip install -r requirements.txt
```

or,

```
python -m pip install -r requirements.txt
```

### 2.2 Running the tool

With the requirements installed, you should be able to run the tool by typing the command (in a terminal window within your python environment):

```
python -m bca_tool_code.tool_main
```

This should create an outputs folder in your project folder (i.e., where you have placed the tool and repository), unless one has already been created, where the results of the run can be found.

Note that the tool has been tested in a Python 3.10 environment.



## 2.3 For help or questions, contact

[sherwood.todd@epa.gov](mailto:sherwood.todd@epa.gov)

## METHODOLOGY

### 3.1 General

The project folder for using the tool should contain an “inputs” folder containing necessary input files and a “bca\_tool\_code” folder containing the Python modules. Optionally, a virtual environment folder may be desirable. When running the tool, the user will be asked to provide a run ID. If a run ID is entered, that run ID will be included in the run-results folder-ID for the given run. Hitting return will use the default run ID. The tool will create an “outputs” folder within the project folder into which all run results will be saved. A timestamp is included in any run-results folder-ID so that new results never overwrite prior results. The user can enter ‘test’ at the run ID prompt. This will send outputs to a ‘test’ folder in the project folder.

The tool first reads inputs and input files. The specific input files to use (i.e., their filenames) must be specified in the Input\_Files.csv file in the “UserEntry.csv” column. The tool then calculates appropriate technology costs, operating costs and emission costs (if selected by the user). Once complete, these are brought together in a set of BCA (benefit-cost analysis) results with those results saved to a run folder within the outputs folder.

Importantly, monetized values in the tool are treated as costs throughout. So a negative cost represents a savings. Also, for the most part, everything is treated in absolute terms. So absolute costs are calculated for each scenario/option/alternative and then deltas are calculated as costs in the action alternative case less costs in the no action, baseline case. As such, higher technology costs in an alternative case than those in the baseline case would result in positive delta costs, or increased costs. Likewise, lower operating costs in an alternative case relative to those in the baseline case would result in negative delta costs, or decreased costs. A decrease in operating costs represents an increase in operating savings.

Note that the calculation of emission impacts is done using the \$/ton estimates included in the CriteriaCostFactors.csv input files. The \$/ton estimates provided in those files are best understood to be the marginal costs associated with the reduction of the individual pollutants as opposed to the absolute costs associated with a ton of each pollutant. As such, the emission “costs” calculated by the tool should not be seen as true costs associated with emissions, but rather the first step in estimating the benefits associated with reductions of those emissions. For that reason, the user must be careful not to consider those as absolute costs, but once compared to the “costs” of another scenario (presumably via calculation of a difference in “costs” between two scenarios) the result can be interpreted as a benefit.

## 3.2 Calculations and Equations

This is not meant to be an exhaustive list of all equations used in the tool, but rather a list of those that are considered to be of most interest. The associated draft Regulatory Impact Analysis (RIA) also contains explanations of calculations made.

### 3.2.1 Learning effects applied to costs

In the criteria air pollutant program calculations, learning effects are applied to direct costs in “steps” which coincide with MY-based cost input columns in the DirectCostInputs\_byRegClass\_byFuelType file. If that input file contains costs for two MY-based steps of implementation, then 2 steps of costs are calculated with step-1 being the first column of cost data and step-2 being the second column. Step-2 costs must be incremental to the step-1 costs entered in the input file. Note that both steps of costs are added to arrive at the direct costs for any given MY. Note also that direct costs are calculated for new vehicles only to represent costs on new vehicle sales.

For step-1 and later direct costs (equations show step-1 occurring in MY2027):

$$DirectCost_{optionID;engine;MY} = \left( \frac{CumulativeSales_{2027+} + Sales_{MY2027} \times SeedVolumeFactor}{Sales_{MY2027} \times (1 + SeedVolumeFactor)} \right)^b \times DirectCost_{optionID;engine;MY2027} \quad (3.1)$$

where,

- $b$  = the learning rate (-0.245 in this analysis but can be changed via the BCA\_General\_Inputs.csv file)
- $DirectCost$  = the direct manufacturing cost absent indirect costs, and where  $DirectCost$  in step-1 (MY2027) is the sum of individual tech costs for step-1 (MY2027) as input via the DirectCostInputs\_byRegClass\_byFuelType file
- $optionID$  = the option considered (i.e., baseline or one of the action alternatives)
- $MY$  = the model year being considered
- $engine$  = a unique regclass-fueltype engine within MOVES
- $CumulativeSales$  = cumulative sales of MY2027 and later engines in model year, MY, of implementation
- $SeedVolumeFactor$  = 0 or greater to represent the number of years of learning already having occurred on a technology

For subsequent steps, e.g., new direct costs implemented in 2030:

$$DirectCost_{optionID;engine;MY} = \left( \frac{CumulativeSales_{2030+} + Sales_{MY2030} \times SeedVolumeFactor}{Sales_{MY2030} \times (1 + SeedVolumeFactor)} \right)^b \times DirectCost_{optionID;engine;MY2030} \quad (3.2)$$

where,

- $CumulativeSales$  = cumulative sales of MY2030 and later engines in model year, MY, of implementation
- $DirectCost$  = marginal direct costs above those calculated for step-1 and later engines (i.e., the sum of individual tech costs for step-2 as input via the DirectCostInputs\_byRegClass\_byFuelType file)

In the Greenhouse Gas Program calculations, learning effects are applied to the technology costs which include both direct and indirect cost. Other than that, they are calculated consistent with what is shown above for criteria air pollutant

costs.

$$\begin{aligned}
 & TechCost_{optionID;vehicle;MY} \\
 &= \left( \frac{CumulativeSales_{2027+} + Sales_{MY2027} \times SeedVolumeFactor}{Sales_{MY2027} \times (1 + SeedVolumeFactor)} \right)^b \times TechCost_{optionID;vehicle;MY2027}
 \end{aligned} \tag{3.3}$$

where,

- $b$  = the learning rate (-0.245 in this analysis but can be changed via the BCA\_General\_Inputs.csv file)
- $TechCost$  = the technology cost inclusive of indirect costs, and where  $TechCost$  in step-1 (MY2027) is from the TechCostInputs\_bySourceType\_byFuelType file
- $optionID$  = the option considered (i.e., baseline or one of the action alternatives)
- $MY$  = the model year being considered
- $vehicle$  = a unique sourcetype-regclass-fueltype vehicle within MOVES
- $CumulativeSales$  = cumulative sales of MY2027 and later vehicles in model year, MY, of implementation
- $SeedVolumeFactor$  = 0 or greater to represent the number of years of learning already having occurred

### 3.2.2 Emission repair costs

The tool calculates emission repair costs associated with changes in warranty and useful life provisions which occur only in the criteria air pollutant program.

#### 3.2.2.1 Direct cost scalers

The direct cost scalers are used to scale the repair cost per mile estimates for engines other than the baseline heavy heavy-duty diesel engine for which the cost per mile inputs apply. In other words, if the cost per mile inputs are \$0.10/mile, and that applies to a heavy heavy-duty diesel engine estimated to cost \$5000, then the cost per mile for that engine after adding \$1000 in new technology would be scaled by \$6000/\$5000 to give a value of \$0.12/mile. Similarly, a light heavy-duty diesel engine costing \$2000 but adding \$500 in new technology would be scaled by \$2500/\$5000 to give a value of \$0.05/mile.

$$DirectCostScalar_{optionID;engine;MY} = \frac{DirectCost_{optionID;engine;MY}}{DirectCost_{Baseline;HHDDDE;MY}} \tag{3.4}$$

where,

- $DirectCost$  = the direct manufacturing cost absent indirect costs
- $optionID$  = the option considered (i.e., baseline or one of the action alternatives)
- $HHDDDE$  = heavy heavy-duty diesel engine regulatory class
- $MY$  = the model year being considered
- $engine$  = a unique regclass-fueltype engine within MOVES

### 3.2.2.2 Estimated warranty & useful life ages

The estimated warranty and useful life ages are used to generate a repair cost per mile curve for each vehicle based on the estimated age when its warranty period will be reached and when its useful life will be reached. These ages differ by sourcetype since sourcetypes accumulate miles at such different rates. Therefore, while a long-haul tractor might reach a 100,000 mile warranty within its first or second year of use, a school bus could take several years to drive that number of miles. If both have a 5 year, 100,000 mile warranty, then the long-haul tractor would have an estimated warranty age of roughly 1 year, while the school bus would have an estimated warranty age of, perhaps, 5 years. The same concepts are true for estimated useful life ages.

$$\begin{aligned} & \text{EstimatedWarrantyAge}_{optionID;vehicle;MY} \\ &= \min(\text{RequiredWarrantyAge}_{optionID;vehicle;MY}, \text{CalculatedWarrantyAge}_{optionID;vehicle;MY}) \end{aligned} \quad (3.5)$$

$$\begin{aligned} & \text{EstimatedUsefulLifeAge}_{optionID;vehicle;MY} \\ &= \min(\text{RequiredUsefulLifeAge}_{optionID;vehicle;MY}, \text{CalculatedUsefulLifeAge}_{optionID;vehicle;MY}) \end{aligned} \quad (3.6)$$

where,

- *RequiredWarrantyAge* = the minimum age required by regulation at which the warranty can end
- *RequiredUsefulLifeAge* = the age required by regulation at which the useful life ends
- *CalculatedWarrantyAge* = the minimum mileage required by regulation at which the warranty can end divided by the “typical” annual miles driven for the given vehicle
- *CalculatedUsefulLifeAge* = the minimum mileage required by regulation at which the useful life can end divided by the “typical” annual miles driven for the given vehicle
- *optionID* = the option considered (i.e., baseline or one of the action alternatives)
- *MY* = the model year being considered
- *vehicle* = a unique sourcetype-regclass-fueltype vehicle within MOVES

Required warranty and useful life miles and ages by optionID/MY/RegClass/FuelType are controlled via input files to the tool (Warranty\_Inputs.csv and UsefulLife\_Inputs.csv, respectively). “Estimated” and “Calculated” ages are calculated by the tool in-code where “Calculated” age uses MOVES sourcetype mileage accumulations. The “typical” annual miles driven is calculated in the tool as the cumulative miles driven divided by the number of years included in the cumulative miles. Because vehicles tend to be driven fewer miles with age, the “typical” annual miles driven decreases with age. The Repair\_and\_Maintenance\_Curve\_Inputs.csv file has a controller for how many years of mileage accumulation to include (typical\_vmt\_thru\_ageID). The default value is 6 which represents 7 years of cumulative miles. Again, a smaller value would result in more “typical” annual miles driven and a lower calculated age, and a larger value would result in fewer “typical” annual miles driven and a higher calculated age.

### 3.2.2.3 Cost per mile by age (for emission-related repairs)

Here the tool estimates the repair cost per mile curve, by age, for each sourcetype-regclass-fueltype vehicle in the analysis. These curves are unique to each type of vehicle and to any options having different warranty and/or useful life provisions.

$$\begin{aligned} & \text{InWarrantyCPM}_{optionID;vehicle;MY} \\ &= \text{FleetAdvantageCPM}_{Year1} \times \text{EmissionRepairShare} \times \text{DirectCostScalar}_{optionID;engine;MY} \end{aligned} \quad (3.7)$$

$$\begin{aligned} & \text{AtUsefulLifeCPM}_{optionID;vehicle;MY} \\ &= \text{FleetAdvantageCPM}_{Year6} \times \text{EmissionRepairShare} \times \text{DirectCostScalar}_{optionID;engine;MY} \end{aligned} \quad (3.8)$$

$$\begin{aligned} & \text{MaxCPM}_{optionID;vehicle;MY} \\ &= \text{FleetAdvantageCPM}_{Year7} \times \text{EmissionRepairShare} \times \text{DirectCostScalar}_{optionID;engine;MY} \end{aligned} \quad (3.9)$$

$$\begin{aligned}
& \text{SlopeCPM}_{\text{optionID};\text{vehicle};\text{MY}} \\
&= \frac{(\text{AtUsefulLifeCPM}_{\text{optionID};\text{vehicle};\text{MY}} - \text{InWarrantyCPM}_{\text{optionID};\text{vehicle};\text{MY}})}{(\text{EstimatedUsefulLifeAge}_{\text{optionID};\text{vehicle};\text{MY}} - \text{EstimatedWarrantyAge}_{\text{optionID};\text{vehicle};\text{MY}})} \quad (3.10)
\end{aligned}$$

where,

- *InWarrantyCPM* = in-warranty emission repair cost per mile for the engine in the given vehicle
- *AtUsefulLifeCPM* = at-useful-life emission repair cost per mile for the engine in the given vehicle
- *MaxCPM* = the maximum emission repair cost per mile for the engine in the given vehicle
- *SlopeCPM* = the cost per mile slope between the estimated warranty age and the estimated useful life age for a given vehicle
- *optionID* = the option considered (i.e, baseline or one of the action alternatives)
- *FleetAdvantageCPMYear1* = first year cost per mile from the Fleet Advantage white paper (2.07 cents/mile in 2018 dollars)
- *FleetAdvantageCPMYear6* = year six cost per mile from the Fleet Advantage white paper (14.56 cents/mile in 2018 dollars)
- *FleetAdvantageCPMYear7* = year seven cost per mile from the Fleet Advantage white paper (19.82 cents/mile in 2018 dollars)
- *EmissionRepairShare* = EPA developed share of Fleet Advantage Maintenance and Repair costs that are emission-related (10.8%)
- *engine* = a unique regclass-fueltype engine for equations (3.7), (3.8) and (3.9)
- *vehicle* = a unique sourcetype-regclass-fueltype vehicle in equation (3.10)

Repair and maintenance cost per mile values—currently based on the Fleet Advantage whitepaper—are controlled via the “Repair\_and\_Maintenance\_Curve\_Inputs.csv” input file to the tool.

For any given optionID/vehicle/MY where vehicle is a unique sourcetype-regclass-fueltype within MOVES, the emission-repair cost per mile (*EmissionRepairCPM*) at any given age would be calculated as:

When  $\text{Age}+1 < \text{EstimatedWarrantyAge}$ :

$$\text{EmissionRepairCPM}_{\text{optionID};\text{vehicle};\text{MY};\text{age}} = \text{InWarrantyCPM}_{\text{optionID};\text{vehicle};\text{MY}} \quad (3.11)$$

When  $\text{EstimatedWarrantyAge} \leq \text{Age}+1 < \text{EstimatedUsefulLifeAge}$ :

$$\begin{aligned}
& \text{EmissionRepairCPM}_{\text{optionID};\text{vehicle};\text{MY};\text{age}} \\
&= \text{SlopeCPM}_{\text{optionID};\text{vehicle};\text{MY}} \times ((\text{Age}_{\text{optionID};\text{vehicle};\text{MY}} + 1) - \text{EstimatedWarrantyAge}_{\text{optionID};\text{vehicle};\text{MY}}) \\
&+ \text{InWarrantyCPM}_{\text{optionID};\text{vehicle};\text{MY}} \quad (3.12)
\end{aligned}$$

When  $\text{Age}+1 = \text{EstimatedUsefulLifeAge}$ :

$$\text{EmissionRepairCPM}_{\text{optionID};\text{vehicle};\text{MY};\text{age}} = \text{AtUsefulLifeCPM}_{\text{optionID};\text{vehicle};\text{MY}} \quad (3.13)$$

Otherwise:

$$\text{EmissionRepairCPM}_{\text{optionID};\text{vehicle};\text{MY};\text{age}} = \text{MaxCPM}_{\text{optionID};\text{vehicle};\text{MY}} \quad (3.14)$$

### 3.2.3 Discounting

#### 3.2.3.1 Present value

$$PV = \frac{AnnualValue_0}{(1 + rate)^{(0+offset)}} + \frac{AnnualValue_1}{(1 + rate)^{(1+offset)}} + \dots + \frac{AnnualValue_n}{(1 + rate)^{(n+offset)}} \quad (3.15)$$

where,

- $PV$  = present value
- $AnnualValue$  = annual costs or annual benefits or annual net of costs and benefits
- $rate$  = discount rate
- $0, 1, \dots, n$  = the period or years of discounting
- $offset$  = controller to set the discounting approach (0 means first costs occur at time=0; 1 means costs occur at time=1)

#### 3.2.3.2 Annualized value

When the present value offset in equation (3.15) equals 0:

$$AV = PV \times \frac{rate \times (1 + rate)^n}{(1 + rate)^{(n+1)} - 1} \quad (3.16)$$

When the present value offset in equation (3.15) equals 1:

$$AV = PV \times \frac{rate \times (1 + rate)^n}{(1 + rate)^n - 1} \quad (3.17)$$

where,

- $AV$  = annualized value of costs or benefits or net of costs and benefits
- $PV$  = present value of costs or benefits or net of costs and benefits
- $rate$  = discount rate
- $n$  = the number of periods over which to annualize the present value

## 3.3 Sensitivites

The BCA\_General\_Inputs file contains several inputs that can be adjusted as indicated within the file. Input values in other files can also be adjusted. It is suggested that the structure of the input files not be changed and that the headers and names within the input files not be changed unless the user is willing to modify the Python code in the event that changes result in errors.

## USEPA-CTI-BCA BCA\_TOOL\_CODE

### 4.1 calc\_deltas module

`calc_deltas.calc_deltas_weighted(settings, dict_for_deltas)`

This function calculates deltas for action alternatives relative to the passed no action alternative specifically for the weighted cost per mile dictionaries.

**Parameters:** settings: The SetInputs class.

dict\_for\_deltas: Dictionary; contains values for calculating deltas.

**Returns:** An updated dictionary containing deltas relative to the no\_action\_alt. OptionIDs (numeric) for the deltas will be the alt\_id followed by the no\_action\_alt.

For example, deltas for optionID=1 relative to optionID=0 would have optionID=10.

**Note:** There is no age\_id or discount rate in the key for the passed weighted dictionaries.

`calc_deltas.calc_deltas(settings, dict_for_deltas)`

This function calculates deltas for action alternatives relative to the no action alternative set via the General Inputs.

**Parameters:** settings: The SetInputs class.

dict\_for\_deltas: Dictionary; contains values for calculating deltas.

**Returns:** An updated dictionary containing deltas relative to the no\_action\_alt. OptionIDs (numeric) for the deltas will be the alt\_id followed by the no\_action\_alt.

For example, deltas for optionID=1 relative to optionID=0 will have optionID=10. OptionNames will also show as 'OptionID=1\_name minus OptionID=0\_name'.

### 4.2 def\_costs module

`def_costs.calc_def_doserate(settings, vehicle)`

**Parameters:** settings: The SetInputs class

vehicle: Tuple; represents a sourcetype\_regclass\_fueltype vehicle.

**Returns:** The DEF dose rate for the passed vehicle based on the DEF dose rate input file.

`def_costs.calc_nox_reduction(settings, vehicle, alt, calendar_year, model_year, totals_dict)`



**Parameters:** settings: The SetInputs class.

vehicle: Tuple; represents an alt\_sourcetype\_regclass\_fueltype vehicle.

alt: Numeric; represents the Alternative or optionID.

calendar\_year: Numeric; represents the calendar year (yearID).

model\_year: Numeric; represents the model year of the passed vehicle.

totals\_dict: Dictionary; provides fleet NOx tons by vehicle.

**Returns:** The NOx reduction for the passed model year vehicle in the given calendar year.

```
def_costs.calc_def_gallons(settings, vehicle, alt, calendar_year, model_year, totals_dict, fuel_arg)
```

**Parameters:** settings: The SetInputs class.

vehicle: Tuple; represents an alt\_sourcetype\_regclass\_fueltype vehicle.

alt: Numeric; represents the Alternative or optionID.

calendar\_year: Numeric; represents the calendar year (yearID).

model\_year: Numeric; represents the model year of the passed vehicle.

totals\_dict: Dictionary; provides gallons (fuel consumption) by vehicle.

fuel\_arg: String; specifies the fuel attribute to use (e.g., "Gallons" or "Gallons\_withTech")

**Returns:** The gallons of DEF consumption for the passed model year vehicle in the given calendar year.

```
def_costs.calc_def_costs(settings, totals_dict, fuel_arg)
```

**Parameters:** settings: The SetInputs class.

totals\_dict: Dictionary; provides fleet DEF consumption by vehicle.

fuel\_arg: String; specifies the fuel attribute to use (e.g., "Gallons" or "Gallons\_withTech")

**Returns:** The passed dictionary updated with costs associated with DEF consumption.

```
def_costs.calc_average_def_costs(totals_dict, averages_dict, vpop_arg)
```

**Parameters:** totals\_dict: Dictionary; provides fleet DEF costs by vehicle.

averages\_dict: Dictionary, into which DEF costs/vehicle will be updated.

vpop\_arg: String; specifies the population attribute to use (e.g., "VPOP" or "VPOP\_withTech")

**Returns:** The passed dictionary updated with costs/mile and costs/vehicle associated with DEF consumption.

## 4.3 direct\_costs module

```
direct_costs.tech_package_cost(settings, unit, alt, cost_step)
```

**Parameters:** settings: The SetInputs class.

unit: Tuple; represents a regclass\_fueltype engine or a sourcetype\_regclass\_fueltype vehicle.

alt: Numeric; represents the Alternative or optionID.

cost\_step: String; represents the model year of implementation of the tech; if standards are implemented in stages (i.e., for MY2027 and then again for MY2031), then these would represent two cost steps, one in '2027' and the other in '2031'.

**Returns:** A single value representing the package direct cost (a summation of individual tech direct costs) for the passed vehicle at the given cost\_step.

`direct_costs.tech_pkg_cost_withlearning(settings, unit, alt, cost_step, sales_arg, cumulative_sales, totals_dict)`

**Parameters:** settings: The SetInputs class.

unit: Tuple; represents a regclass\_fueltype engine or a sourcetype\_regclass\_fueltype vehicle.

alt: The alternative or option ID.

cost\_step: String; represents the model year of implementation in case standards are implemented in stages then these would represent multiple cost steps.

sales\_arg: String; specifies the sales attribute to use (e.g., "VPOP" or "VPOP\_withTech")

cumulative\_sales: Numeric; represents cumulative sales of unit since cost\_step.

totals\_dict: A dictionary containing sales (sales\_arg) of units by model year.

**Returns:** The package cost with learning applied for the passed unit in the given model year and associated with the given cost\_step.

`direct_costs.calc_yoy_costs_per_step(settings, totals_dict, sales_arg, program)`

**Parameters:** settings: The SetInputs class.

totals\_dict: Dictionary; provides sales of units by model year; this will be faster if age\_id > 0 is scrubbed out first.

sales\_arg: String; specifies the sales attribute to use (e.g., "VPOP" or "VPOP\_withTech").

program: String; the program identifier (e.g., 'CAP' or 'GHG').

**Returns:** A dictionary containing the package cost and cumulative sales used to calculate that package cost (learning effects depend on cumulative sales) for the passed unit in the given model year and complying with the standards set in the given cost step.

`direct_costs.calc_per_vch_direct_costs(yoy_costs_per_step_dict, cost_steps, averages_dict, program)`

**Parameters:** yoy\_costs\_per\_step\_dict: Dictionary; contains the package cost and cumulative sales used to calculate that package cost (learning effects depend on cumulative sales) for the passed unit in the given model year and complying with the standards set in the given cost step.

cost\_steps: List; provides the cost steps (as strings) associated with the direct costs being calculated.

averages\_dict: Dictionary; into which tech package direct costs/vehicle will be updated.

program: String; the program identifier (i.e., 'CAP' or 'GHG').

**Returns:** The averages\_dict dictionary updated with tech package costs/vehicle.

`direct_costs.calc_direct_costs(totals_dict, averages_dict, sales_arg, program)`

**Parameters:** totals\_dict: Dictionary; into which tech package direct costs will be updated.

averages\_dict: Dictionary; contains tech package direct costs/vehicle.

`sales_arg`: String; specifies the sales attribute to use (e.g., “VPOP” or “VPOP\_withTech”)

`program`: String; the program identifier (i.e., ‘CAP’ or ‘GHG’).

**Returns:** The `totals_dict` dictionary updated with tech package direct costs (package cost \* sales).

## 4.4 discounting module

`discounting.discount_values(settings, dict_of_values, program, arg)`

The discount function determines metrics appropriate for discounting (those contained in `dict_of_values`) and does the discounting calculation to a given year and point within that year.

**Parameters:** `settings`: The `SetInputs` class.

`dict_of_values`: Dictionary; provides values to be discounted with keys consisting of vehicle, model\_year, age\_id and discount rate.

`program`: String; indicates what program is being passed. `arg`: String; indicates whether totals or averages are being discounted.

**Returns:** The passed dictionary with new key, value pairs where keys stipulate the discount rate and monetized values are discounted at the same rate as the discount rate of the input stream of values.

**Note:** The `costs_start` entry of the `BCA_General_Inputs` file should be set to ‘start-year’ or ‘end-year’, where start-year represents costs starting at time  $t=0$  (i.e., first year costs are undiscounted), and end-year represents costs starting at time  $t=1$  (i.e., first year costs are discounted).

## 4.5 emission\_costs module

`emission_costs.get_criteria_cost_factors(settings, calendar_year)`

**Parameters:** `settings`: The `SetInputs` class.

`calendar_year`: Numeric; the calendar year for which emission cost factors are needed.

**Returns:** Six values - the PM25, NOx and SO2 emission cost factors (dollars/ton) for each of two different mortality estimates and each of two different discount rates.

**Note:** Note that the `BCA_General_Inputs` file contains a toggle to stipulate whether to estimate emission (pollution) costs or not. This function is called only if that toggle is set to ‘Y’ (yes). The default setting is ‘N’ (no).

`emission_costs.calc_criteria_emission_costs(settings, totals_dict)`

**Parameters:** `settings`: The `SetInputs` class.

`totals_dict`: Dictionary; into which emission costs will be updated.

**Returns:** The `totals_dict` dictionary updated with emission costs (\$/ton \* inventory tons).

## 4.6 figures module

figures.py

Contains the CreateFigures class.

**class** figures.**CreateFigures**(*df, units, destination, program*)

Bases: object

**line\_chart\_args\_by\_option**(*dr, alt\_name, year\_min, year\_max, \*args*)

This method generates a chart showing passed arguments under the given alternative.

**Parameters:** dr: Numeric; the discount rate of the data to be charted.

alt\_name: String; the OptionName of the data to be charted.

year\_min: Numeric; the minimum calendar year of data to be charted.

year\_max: Numeric; the maximum calendar year of data to be charted.

args: String(s); the data attributes to be charted.

**Returns:** A single chart saved to the destination folder.

**line\_chart\_arg\_by\_options**(*dr, alt\_names, year\_min, year\_max, arg*)

This method generates a chart showing the passed argument under each of the passed alternatives.

**Parameters:** dr: Numeric; the discount rate of the data to be charted.

alt\_names: List; contains the OptionNames for which to chart data.

year\_min: Numeric; the minimum calendar year of data to be charted.

year\_max: Numeric; the maximum calendar year of data to be charted.

arg: String; the single data attribute to be charted.

**Returns:** A single chart saved to the destination folder.

**create\_figures**(*args*)

This function is called by tool\_main and then controls the generation of charts by the ChartFigures class.

**Parameters:** args: List; attributes to include in figures.

**Returns:** Charts are saved to the path\_for\_save folder by the ChartFigures class and this method returns to tool\_main.

## 4.7 fleet\_totals\_dict module

fleet\_totals\_dict.**add\_keys\_for\_discounting**(*input\_dict, \*rates*)

**Parameters:** input\_dict: Dictionary; into which new keys will be added that provide room for discounting data.

rates: Numeric; the discount rate keys to add.

**Returns:** The passed dictionary with new keys added.

**class** fleet\_totals\_dict.**FleetTotals**(*fleet\_dict*)

Bases: object

A FleetTotals object contains annual totals for vehicles by model year and by calendar year.

**Parameters:** fleet\_dict: Dictionary; contains fleet data, totals by model year and calendar year.

**create\_new\_attributes**(*calc\_cap\_pollution, calc\_ghg\_pollution*)

**Parameters:** calc\_cap\_pollution: True or None.

calc\_ghg\_pollution: True or None.

**Returns:** A list of new attributes to be calculated and provided in output files.

**create\_fleet\_totals\_dict**(*settings, fleet\_df*)

This method creates a dictionary of fleet total values and adds a discount rate element to the key.

**Parameters:** settings: The SetInputs class.

fleet\_df: DataFrame; the project fleet.

**Returns:** A dictionary of the fleet having keys equal to ((vehicle), modelYearID, ageID, discount\_rate) where vehicle is a tuple representing an alt\_sourcetype\_regclass\_fueltype vehicle, and values representing totals for each key over time.

**create\_regclass\_sales\_dict**(*fleet\_df*)

This method simply generates sales by regclass via Pandas which is faster than summing via dictionary.

**Parameters:** fleet\_df: DataFrame; the project fleet.

**Returns:** A dictionary of the fleet having keys equal to ((unit), alt, modelYearID) where unit is a tuple representing a regclass\_fueltype, and values representing sales (sales=VPOP at ageID=0) for each key by model year.

**create\_sourcetype\_sales\_dict**(*fleet\_df*)

This method simply generates sales by sourcetype via Pandas which is faster than summing via dictionary.

**Parameters:** fleet\_df: DataFrame; the project fleet.

**Returns:** A dictionary of the fleet having keys equal to ((unit), alt, modelYearID) where unit is a tuple representing a sourcetype\_regclass\_fueltype, and values representing sales (sales=VPOP at ageID=0) for each key by model year.

**calc\_unit\_sales**(*unit, alt, model\_year, sales\_arg*)

**Parameters:** unit: Tuple; represents a regclass-fueltype engine or sourcetype-regclass-fueltype vehicle.

alt: Numeric; represents the Alternative or optionID.

model\_year: Numeric; represents the model year of the passed unit.

sales\_arg: String; represents the sales attribute to use.

**Returns:** A single sales value (Numeric, i.e., sales\_arg value) for the given unit, model\_year, alt.

**Note:** DiscountRate is set to zero since sales numbers will not change with discount rate.

**calc\_unit\_cumulative\_sales**(*unit, alt, start\_model\_year, end\_model\_year, sales\_arg*)

**Parameters:** unit: Tuple; represents a regclass-fueltype engine or sourcetype-regclass-fueltype vehicle.

alt: Numeric; represents the Alternative or optionID.

start\_model\_year: Numeric; represents the initial model year of sales to include.

end\_model\_year: Numeric; represents the final model year of sales to include (e.g., the unit's model year).

sales\_arg: String; represents the sales attribute to use.

**Returns:** A single cumulative sales value (Numeric, i.e., sales\_arg value) for the given unit, model\_year, alt.

**Note:** DiscountRate is set to zero since sales numbers will not change with discount rate.

**update\_dict**(key, input\_dict)

**Parameters:** key: Tuple; the key of the dictionary instance.

input\_dict: Dictionary; represents the attribute-value pairs to be updated.

**Returns:** The dictionary instance with each attribute updated with the appropriate value.

**get\_attribute\_value**(key, attribute)

**Parameters:** key: Tuple; the key of the dictionary instance.

attribute: String; represents the attribute to be updated.

**Returns:** The value of 'attribute' within the dictionary instance.

## 4.8 fleet\_averages\_dict module

**class** fleet\_averages\_dict.FleetAverages(fleet\_dict)

Bases: object

A FleetAverages object contains annual averages for vehicles by model year and by calendar year.

**Parameters:** fleet\_dict: Dictionary; contains fleet data, averages by model year and calendar year.

**create\_new\_attributes**()

**Returns:** A list of new attributes to be calculated and provided in output files.

**create\_fleet\_averages\_dict**(settings, fleet\_df)

This function creates a dictionary of fleet average values and adds a discount rate element to the key. It also calculates an average annual VMT/vehicle and a cumulative annual average VMT/vehicle.

**Parameters:** settings: The SetInputs class.

fleet\_df: DataFrame; the project fleet DataFrame.

**Returns:** A dictionary of the fleet having keys equal to ((vehicle), modelYearID, ageID, discount\_rate) where vehicle is a tuple representing an alt\_sourcetype\_regclass\_fueltype vehicle, and values representing per vehicle or per mile averages for each key over time.

**update\_dict**(key, input\_dict)

**Parameters:** key: Tuple; the key of the dictionary instance.

input\_dict: Dictionary; represents the attribute-value pairs to be updated.

**Returns:** The dictionary instance with 'attribute' updated with 'value.'

**get\_attribute\_value**(key, attribute)

**Parameters:** key: Tuple; the key of the dictionary instance.

attribute: String; represents the attribute to be updated.

**Returns:** The value of 'attribute' within the dictionary instance.

**static** `calc_per_veh_cumulative_vmt(fleet_dict)`

This function calculates cumulative average VMT/vehicle year-over-year for use in estimating a typical VMT per year and for estimating emission repair costs.

**Parameters:** `fleet_dict`: Dictionary; represents the dictionary instance.

**Returns:** The dictionary instance updated with cumulative annual average VMT/vehicle.

**Note:** VMT does not differ across options.

## 4.9 fuel\_costs module

`fuel_costs.get_orvr_adjustment(settings, vehicle, alt, program)`

**Parameters:** `settings`: The SetInputs class.

`vehicle`: Tuple; represents a `sourcetype_regclass_fueltype` vehicle.

`alt`: Numeric; the Alternative or option ID.

`program`: String; represents which program is being run, CAP or GHG.

**Returns:** A single value representing the milliliter per gram adjustment to be applied to total hydrocarbon emission reductions to estimate the gallons of fuel saved.

`fuel_costs.calc_thc_reduction(settings, vehicle, alt, calendar_year, model_year, totals_dict)`

**Parameters:** `settings`: The SetInputs class.

`vehicle`: Tuple; represents a `sourcetype_regclass_fueltype` vehicle.

`alt`: Numeric; the Alternative or option ID.

`calendar_year`: Numeric; the calendar year.

`model_year`: Numeric; the model year of the passed vehicle.

`totals_dict`: Dictionary; provides fleet total hydrocarbon (THC) tons by vehicle.

**Returns:** A single THC reduction for the given model year vehicle in the given year.

`fuel_costs.calc_captured_gallons(settings, vehicle, alt, calendar_year, model_year, totals_dict, program)`

**Parameters:** `settings`: The SetInputs class.

`vehicle`: Tuple; represents a `sourcetype_regclass_fueltype` vehicle.

`alt`: Numeric; the Alternative or option ID.

`calendar_year`: Numeric; the calendar year.

`model_year`: Numeric; the model year of the passed vehicle.

`totals_dict`: Dictionary; provides fleet total hydrocarbon (THC) tons by vehicle. `program`: String; represents which program is being run, CAP or GHG.

**Returns:** The gallons captured by ORVR that would have otherwise evaporated during refueling.

`fuel_costs.calc_fuel_costs(settings, totals_dict, fuel_arg, program)`

**Parameters:** settings: The SetInputs class.

totals\_dict: Dictionary; provides the fleet Gallons consumed by all vehicles.

fuel\_arg: String; specifies the fuel attribute to use (e.g., “Gallons” or “Gallons\_withTech”)

program: String; represents which program is being run, CAP or GHG.

**Returns:** The passed dictionary updated to reflect fuel consumption (Gallons) adjusted to account for the fuel saved in association with ORVR. The dictionary is also updated to include the fuel costs associated with the gallons consumed (Gallons \* \$/gallon fuel).

**Note:** Note that gallons of fuel captured are not included in the MOVES runs that serve as the input fleet data for the tool although the inventory impacts are included in the MOVES runs.

`fuel_costs.calc_average_fuel_costs(totals_dict, averages_dict, vpop_arg, vmt_arg)`

**Parameters:** totals\_dict: Dictionary; provides fleet fuel costs for all vehicles.

averages\_dict: Dictionary; the destination for fuel costs/vehicle and costs/mile results.

vpop\_arg: String; specifies the population attribute to use (e.g., “VPOP” or “VPOP\_withTech”)

vmt\_arg: String; specifies the VMT attribute to use (e.g., “VMT” or “VMT\_withTech”)

**Returns:** The passed averages\_dict updated to include fuel costs/vehicle and costs/mile.

## 4.10 general\_functions module

`general_functions.inputs_filenames(input_files_pathlist)`

**Parameters:** input\_files\_pathlist: List; those input files that are specified in the Input\_Files.csv file contained in the inputs folder.

**Returns:** A list of input file full paths - these will be copied directly to the output folder so that inputs and outputs end up bundled together in the output folder associated with the given run.

`general_functions.reshape_df(df, value_variable_list, cols_to_melt, melted_header, new_column_name)`

**Parameters:** df: The DataFrame to melt.

value\_variable\_list: Column(s) list to use as identifier variables.

cols\_to\_melt: Column(s) list of columns to pivot (melt).

melted\_header: The header for the column to be populated with the cols\_to\_melt list.

new\_column\_name: Name to use for the ‘Value’ column.

**Returns:** A new DataFrame in long and narrow shape rather than the passed short and wide shape.

**Note:** This function is not being used.

`general_functions.convert_dollars_to_analysis_basis(df, deflators, dollar_basis, *args)`

This function converts dollars into a consistent dollar basis as set via the General Inputs file.

**Parameters:** df: DataFrame; contains the monetized values and their associated input cost basis.

deflators: Dictionary; provides GDP deflators for use in adjusting monetized values throughout the tool into a consistent dollar basis.

dollar\_basis: Numeric; the dollar basis to be used throughout the analysis as set via the General Inputs file.



args: String(s); the attributes within the passed df to be adjusted into 'dollar\_basis' dollars.

**Returns:** The passed DataFrame will all args adjusted into dollar\_basis dollars.

`general_functions.round_metrics(df, metrics, round_by)`

**Parameters:** df: DataFrame containing data to be rounded.

metrics: List of metrics within the passed DataFrame for which rounding is requested.

round\_by: A value that sets the level of rounding.

**Returns:** The passed DataFrame with 'metrics' rounded by 'round\_by'.

**Note:** This function is not being used.

`general_functions.round_sig(df, divisor=1, sig=0, *args)`

**Parameters:** df: The DataFrame containing data to be expressed in 'sig' significant digits.

divisor: The divisor to use in calculating results.

sig: The number of significant digits to use for results.

args: The arguments to be expressed in 'sig' significant digits and in 'divisor' units.

**Returns:** The passed DataFrame with args expressed in 'sig' significant digits and in 'divisor' units.

**Note:** This function is not being used.

`general_functions.get_file_datetime(list_of_files)`

**Parameters:** list\_of\_files: List; the files for which datetimes are required.

**Returns:** A DataFrame of input files (full path) and corresponding datetimes (date stamps) for those files.

`general_functions.read_input_files(path, input_file, usecols=None, index_col=None, skiprows=None, reset_index=False)`

**Parameters:** path: String; the path to input files.

input\_file: String; the file (filename) to read.

usecols: List; the columns to used in the returned DataFrame.

index\_col: Numeric; the column to use as the index column of the returned DataFrame.

skiprows: Numeric; the number of rows to skip when reading the file.

reset\_index: Boolean; True resets index, False does not.

**Returns:** A DataFrame of the desired data from the passed input file.

`general_functions.get_common_metrics(df_left, df_right, ignore=None)`

This function simply finds common metrics between 2 DataFrames being merged to ensure a safe merge.

**Parameters:** df\_left: The left DataFrame being merged.

df\_right: The right DataFrame being merged.

ignore: Any columns (arguments) to ignore when finding common metrics.

**Returns:** A DataFrame merged on the common arguments (less any ignored arguments).

**Note:** This function is not being used.

`general_functions.save_dict_to_csv(dict_to_save, save_path, row_header=None, *args)`

**Parameters:** dict\_to\_save: Dictionary; a dictionary having a tuple of args as keys.

save\_path: Path object; the path for saving the passed CSV.

row\_header: List; the column names to use as the row header for the preferred structure of the output file.

args: String(s); the attributes contained in the tuple key - these will be pulled out and named according to the passed arguments.

**Returns:** A CSV file with individual key elements split out into columns with args as names.

## 4.11 get\_context\_data module

`class get_context_data.GetFuelPrices(input_file, aeo_case, id_col, *fuels)`

Bases: object

The GetFuelPrices class grabs the appropriate fuel prices from the aeo folder, cleans up some naming and creates a fuel\_prices DataFrame for use in operating costs.

**Parameters:** input\_file: String; the file containing fuel price data.

aeo\_case: String; from the General Inputs sheet - the AEO fuel case to use (a CSV of fuel prices must exist in the aeo directory).

id\_col: String; the column name where id data can be found.

fuels: String(s); the AEO descriptor for the fuel prices needed in the project (e.g., Motor Gasoline, Diesel).

**Note:** This class assumes a file structured like those published by EIA in the Annual Energy Outlook.

`aeo_dollars()`

**Returns:** An integer value representing the dollar basis of the AEO report.

`select_aeo_table_rows(df_source, row)`

**Parameters:** df\_source: DataFrame; contains the AEO fuel prices.

row: String; the specific row to select.

**Returns:** A DataFrame of the specific fuel price row.

`row_dict(fuel)`

**Parameters:** fuel: String; the fuel (gasoline/diesel).

**Returns:** A dictionary of fuel prices.

`melt_df(df, value_name)`

**Parameters:** df: DataFrame; the fuel prices to melt.

value\_name: String; the name of the melted values.

**Returns:** A DataFrame of melted value\_name data by year.

`get_prices()`

**Returns:** A DataFrame of fuel prices for the given AEO case. Note that CNG prices are set equivalent to gasoline prices.

**class** `get_context_data.GetDeflators(input_file, id_col, id_value)`

Bases: `object`

The GetDeflators class returns the GDP Implicit Price Deflators for use in adjusting monetized values to a consistent cost basis.

**Parameters:** `input_file`: String; the file containing price deflator data.

`id_col`: String; the column name where id data can be found.

`id_value`: the value within `id_col` to return.

**Note:** This class assumes a file structured like those published by the Bureau of Economic Analysis.

**deflator\_df()**

**Returns:** A DataFrame consisting of only the data for the given AEO case; the name of the AEO case is also removed from the 'full name' column entries.

**melt\_df(value\_name, drop\_col=None)**

**Parameters:** `value_name`: String; the name for the resultant data column.

`drop_col`: String; the name of any columns to be dropped after melt.

**Returns:** The melted DataFrame with a column of data named `value_name`.

**calc\_adjustment\_factors(dollar\_basis)**

**Parameters:** `dollar_basis`: Numeric; the dollar basis for the analysis set via the General Inputs sheet.

**Returns:** A dictionary of deflators and `adjustment_factors` to apply to monetized values to put them all on a consistent dollar basis.

## 4.12 indirect\_costs module

**indirect\_costs.calc\_project\_markup\_value(settings, unit, alt, markup\_factor\_name, model\_year)**

This function calculates the project markup value for the `markup_factor` (Warranty, RnD, Other, Profit) passed.

**Parameters:** `settings`: The SetInputs classs.

`unit`: Tuple; represents a `regclass_fueltype` engine or a `sourcetype_regclass_fueltype` vehicle.

`alt`: Numeric; The Alternative or option ID.

`markup_factor_name`: String; represents the name of the project markup factor value to return (warranty, r and d, other, etc.).

`model_year`: Numeric; the model year of the passed unit.

**Returns:** A single markup factor value to be used in the project having been adjusted in accordance with the proposed warranty and useful life changes and the Absolute/Relative scaling entries.

**Note:** The project markup factor differs from the input markup factors by scaling where that scaling is done based on the "Absolute" or "Relative" entries in the input file and by the scaling metric (Miles or Age) entries of the warranty/useful life input files. Whether Miles or Age is used is set via the `BCA_General_Inputs` file.

**indirect\_costs.calc\_per\_veh\_indirect\_costs(settings, averages\_dict)**

**Parameters:** settings: The SetInputs class.

averages\_dict: Dictionary; contains tech package direct costs/vehicle.

**Returns:** The averages\_dict dictionary updated with indirect costs associated with each markup value along with the summation of those individual indirect costs as “IndirectCost\_AvgPerVeh.”

`indirect_costs.calc_indirect_costs(settings, totals_dict, averages_dict, sales_arg)`

**Parameters:** settings: The SetInputs class.

totals\_dict: Dictionary; contains sales data (e.g., sales\_arg at age=0).

averages\_dict: Dictionary; contains individual indirect costs per vehicle.

sales\_arg: String; specifies the sales attribute to use (e.g., “VPOP” or “VPOP\_withTech”)

**Returns:** The totals\_dict dictionary updated with total indirect costs for each individual indirect cost property and a summation of those.

## 4.13 project\_dicts module

**class** project\_dicts.InputFileDict(input\_dict)

Bases: object

The InputFileDict class object contains a dictionary of the data provided via the input files for use throughout the tool.

**create\_project\_dict**(input\_df, \*args)

**Parameters:** input\_df: DataFrame; contains data from the applicable input file.

args: String(s); attributes to include in the returned dictionary key.

**Returns:** The passed DataFrame as a dictionary with keys consisting of the passed args.

**get\_attribute\_value**(key, attribute)

**Parameters:** key: Tuple; the key of the dictionary instance.

attribute: String; represents the attribute to be updated.

**Returns:** The value of ‘attribute’ within the dictionary instance.

## 4.14 project\_fleet module

`project_fleet.create_fleet_df(settings, input_df, options_dict, adj_dict, *args_with_tech)`

**Parameters:** settings: The SetInputs class.

input\_df: DataFrame; the raw fleet input data (e.g., from MOVES).

options\_dict: Dictionary; provides the option IDs and names of options being run.

adj\_dict: Dictionary; provides any adjustments to be made to the data contained in input\_df.

args\_with\_tech: String(s); the attributes to be adjusted.

**Returns:** A DataFrame of the MOVES inputs with necessary MOVES adjustments made according to the MOVES adjustments input file. The DataFrame will also add optionID/sourceTypeID/regClassID/fuelTypeID names and will use only those options included in the options.csv input file.

`project_fleet.regclass_vehicles(fleet_df)`

**Parameters:** fleet\_df: A DataFrame of the project fleet.

**Returns:** A series of unique vehicles where a vehicle is a ((regClass, fuelType), alt) vehicle.

`project_fleet.sourcetype_vehicles(fleet_df)`

**Parameters:** fleet\_df: A DataFrame of the project fleet.

**Returns:** A series of unique vehicles where a vehicle is a ((sourceType, regClass, fuelType), alt) vehicle.

## 4.15 repair\_costs module

`repair_costs.calc_typical_vmt_per_year(settings, vehicle, alt, model_year, averages_dict)`

This function calculates a typical annual VMT/vehicle over a set number of years as set via the General Inputs workbook. This typical annual VMT/vehicle can then be used to estimate the ages at which warranty and useful life will be reached. When insufficient years are available – e.g., if the typical\_vmt\_thru\_ageID is set to >5 years and the given vehicle is a MY2041 vintage vehicle and the fleet input file contains data only thru CY2045, then insufficient data exist to calculate the typical VMT for that vehicle – the typical VMT for that vehicle will be set equal to the last prior MY vintage for which sufficient data were present.

**Parameters:** settings: The SetInputs class.

vehicle: Tuple; represents a sourcetype\_regclass\_fueltype vehicle.

alt: Numeric; the Alternative or option ID.

model\_year: Numeric; the model year of the passed vehicle.

averages\_dict: Dictionary; contains cumulative annual average VMT/vehicle.

**Returns:** A single typical annual VMT/veh value for the passed vehicle of the given model year.

`repair_costs.calc_estimated_age(settings, vehicle, alt, model_year, identifier, typical_vmt, estimated_ages_dict)`

**Parameters:** settings: The SetInputs class.

vehicle: Tuple; represents a sourcetype\_regclass\_fueltype vehicle.

alt: Numeric; the Alternative or option ID.

model\_year: Numeric; the model year of the passed vehicle.

identifier: String; the identifier of the age being estimated (i.e., 'Warranty' or 'Usefullife').

typical\_vmt: Numeric; the typical annual VMT/vehicle over a set number of years as set via the General Inputs workbook (see calc\_typical\_vmt\_per\_year function).

estimated\_ages\_dict: Dictionary in which to collect estimated ages to be included in the outputs for the given run.

**Returns:** An integer representing the age at which the identifier will be reached for the passed vehicle/model year.

A dictionary that tracks those ages for all vehicles.

`repair_costs.calc_emission_repair_costs_per_mile(settings, averages_dict)`

**Parameters:** settings: The SetInputs class.

averages\_dict: Dictionary; contains tech package direct costs/vehicle and cumulative annual average VMT/vehicle.

**Returns:** The averages\_dict dictionary updated to include emission repair costs/mile for each dictionary key.

A repair cost/mile dictionary containing details used in the calculation of repair cost/mile and which is then written to an output file for the given run.

An 'estimated ages' dictionary containing details behind the calculations and which is then written to an output file for the given run.

`repair_costs.calc_per_veh_emission_repair_costs(averages_dict)`

**Parameters:** averages\_dict: Dictionary; contains annual emission repair costs/mile.

**Returns:** The passed dictionary updated with annual emission repair costs/vehicle for each dictionary key.

`repair_costs.calc_emission_repair_costs(totals_dict, averages_dict, vpop_arg)`

**Parameters:** totals\_dict: Dictionary; contains annual vehicle populations (VPOP).

averages\_dict: Dictionary; contains annual average emission repair costs/mile.

vpop\_arg: String; specifies the population attribute to use (e.g., "VPOP" or "VPOP\_withTech")

**Returns:** The totals\_dict dictionary updated with annual emission repair costs for all vehicles.

## 4.16 sum\_by\_vehicle module

`sum_by_vehicle.calc_sum_of_costs(dict_to_sum, name_of_sum, *args)`

**Parameters::** dict\_to\_sum: Dictionary; contains the parameters to be summed.

name\_of\_sum: String; used to identify the sum being done.

args: String(s); the attributes to be summed.

**Returns:** The passed dictionary updated to include a new 'name\_of\_sum' parameter that sums the passed args for each dictionary key.

## 4.17 tech\_costs module

`tech_costs.calc_per_vch_tech_costs(averages_dict)`

**Parameters::** averages\_dict: Dictionary; contains average direct and indirect costs per vehicle.

**Returns:** The averages\_dict dictionary updated with average tech costs per vehicle (direct plus indirect).

**Note:** Direct and indirect costs apply only for ageID=0 (i.e., new sales).

`tech_costs.calc_tech_costs(totals_dict, averages_dict, sales_arg)`

**Parameters::** totals\_dict: Dictionary; contains vehicle population (VPOP) data.

averages\_dict: Dictionary; contains average tech costs per vehicle. sales\_arg: String; specifies the sales attribute to use (e.g., "VPOP" or "VPOP\_withTech")

**Returns:** The totals\_dict dictionary updated with annual technology costs for all vehicles.

## 4.18 tool\_main module

`cti_bca_tool.tool_main.py`

This is the main module of the tool.

`tool_main.main()`

**Returns:** The results of the current run of the tool.

## 4.19 vehicle module

`class vehicle.Vehicle(id=None)`

Bases: object

Define vehicle attribute names for sourceTypeID, regClassID, fuelTypeID.

**Parameters::** id: The associated ID from the MOVES input file.

**Returns:** Source type name, Regclass name, Fuel type name.

`fueltype_name()`

**Returns:** The fuel type name for the passed ID.

`regclass_name()`

**Returns:** The regclass name for the passed ID.

`sourcetype_name()`

**Returns:** The source type name for the passed ID.

`static vehicle_name(settings, options_dict, dict_of_vehicles)`

**Parameters:** settings: The SetInputs class.

options\_dict: Dictionary; provides the option ID numbers and associated names.

dict\_of\_vehicles: Dictionary; contains keys of vehicle tuples.

**Returns:** The passed dictionary with new attributes identifying the vehicle based on the vehicle tuples (keys).

**static option\_name**(settings, options\_dict, dict\_of\_vehicles)

**Parameters:** settings: The SetInputs class. options\_dict: Dictionary; provides the option ID numbers and associated names.

dict\_of\_vehicles: Dictionary; contains keys of vehicle tuples.

**Returns:** The passed dictionary with new attributes identifying the option name.

## 4.20 weighted\_results module

weighted\_results.**create\_weighted\_cost\_dict**(settings, averages\_dict, arg\_to\_weight, arg\_to\_weight\_by)

This function weights 'arg\_to\_weight' attributes by the 'arg\_to\_weight\_by' attribute.

**Parameters::** settings: The SetInputs class.

averages\_dict: Dictionary; contains fleet average data (e.g., miles/year, cost/year, cost/mile).

arg\_to\_weight: String; the attribute to be weighted by the arg\_to\_weight\_by argument.

arg\_to\_weight\_by: String; the argument to weight by.

**Returns:** A dictionary of arguments weighted by the weight\_by argument.

**Note:** The weighting is limited by the number of years (ages) to be included which is set in the general inputs file. The weighting is also limited to model years for which sufficient data exists to include all of those ages. For example, if the maximum calendar year included in the input data is 2045, and the maximum numbers of ages of data to include for each model year is 9 (which would be 10 years of age since year 1 is age 0) then the maximum model year included will be 2035.



## DISTRIBUTION AND SUPPORT POLICY FOR EPA SOFTWARE

### 5.1 Copyright Status

The United States (U.S.) Government retains a non-exclusive, royalty-free license to publish or reproduce these software products and associated documents, or allow others to do so, for U.S. Government purposes. These software products and documents may be freely distributed and used for non-commercial, scientific, and/or educational purposes. Commercial use of these software products and documents may be protected under the U.S. and Foreign Copyright Laws. Individual documents on this server may have different copyright conditions, and that will be noted in those documents.

### 5.2 Disclaimer of Liability

With respect to this multimedia system of HTML pages and the EPA software products and their documentation, neither the U.S. Government nor any of their employees, makes any warranty, express or implied, including the warranties of merchantability and fitness for a particular purpose, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights.

### 5.3 Disclaimer of Software Installation/Application

Execution of any EPA installation program, and modification to system configuration files must be made at the user's own risk. Neither the U.S. EPA nor the program author(s) can assume responsibility for program modification, content, output, interpretation, or usage.

EPA installation programs have been extensively tested and verified. However, as for all complex software, these programs may not be completely free of errors and may not be applicable for all cases. In no event will the U.S. EPA be liable for direct, indirect, special, incidental, or consequential damages arising out of the use of the programs and/or associated documentation.

## INDICES AND TABLES

- `genindex`
- `modindex`
- `search`

## PYTHON MODULE INDEX

### C

`calc_deltas`, 13

### d

`def_costs`, 13

`direct_costs`, 14

`discounting`, 16

### e

`emission_costs`, 16

### f

`figures`, 17

`fleet_averages_dict`, 19

`fleet_totals_dict`, 17

`fuel_costs`, 20

### g

`general_functions`, 21

`get_context_data`, 23

### i

`indirect_costs`, 24

### p

`project_dicts`, 25

`project_fleet`, 25

### r

`repair_costs`, 26

### s

`sum_by_vehicle`, 27

### t

`tech_costs`, 28

`tool_main`, 28

### v

`vehicle`, 28

### w

`weighted_results`, 29

## A

`add_keys_for_discounting()` (in module `fleet_totals_dict`), 17  
`aeo_dollars()` (`get_context_data.GetFuelPrices` method), 23

## C

`calc_adjustment_factors()` (`get_context_data.GetDeflators` method), 24  
`calc_average_def_costs()` (in module `def_costs`), 14  
`calc_average_fuel_costs()` (in module `fuel_costs`), 21  
`calc_captured_gallons()` (in module `fuel_costs`), 20  
`calc_criteria_emission_costs()` (in module `emission_costs`), 16  
`calc_def_costs()` (in module `def_costs`), 14  
`calc_def_doserate()` (in module `def_costs`), 13  
`calc_def_gallons()` (in module `def_costs`), 14  
`calc_deltas` module, 13  
`calc_deltas()` (in module `calc_deltas`), 13  
`calc_deltas_weighted()` (in module `calc_deltas`), 13  
`calc_direct_costs()` (in module `direct_costs`), 15  
`calc_emission_repair_costs()` (in module `repair_costs`), 27  
`calc_emission_repair_costs_per_mile()` (in module `repair_costs`), 27  
`calc_estimated_age()` (in module `repair_costs`), 26  
`calc_fuel_costs()` (in module `fuel_costs`), 20  
`calc_indirect_costs()` (in module `indirect_costs`), 25  
`calc_nox_reduction()` (in module `def_costs`), 13  
`calc_per_veh_cumulative_vmt()` (`fleet_averages_dict.FleetAverages` static method), 20  
`calc_per_veh_direct_costs()` (in module `direct_costs`), 15  
`calc_per_veh_emission_repair_costs()` (in module `repair_costs`), 27  
`calc_per_veh_indirect_costs()` (in module `indirect_costs`), 24

`calc_per_veh_tech_costs()` (in module `tech_costs`), 28  
`calc_project_markup_value()` (in module `indirect_costs`), 24  
`calc_sum_of_costs()` (in module `sum_by_vehicle`), 27  
`calc_tech_costs()` (in module `tech_costs`), 28  
`calc_thc_reduction()` (in module `fuel_costs`), 20  
`calc_typical_vmt_per_year()` (in module `repair_costs`), 26  
`calc_unit_cumulative_sales()` (`fleet_totals_dict.FleetTotals` method), 18  
`calc_unit_sales()` (`fleet_totals_dict.FleetTotals` method), 18  
`calc_yoy_costs_per_step()` (in module `direct_costs`), 15  
`convert_dollars_to_analysis_basis()` (in module `general_functions`), 21  
`create_figures()` (`figures.CreateFigures` method), 17  
`create_fleet_averages_dict()` (`fleet_averages_dict.FleetAverages` method), 19  
`create_fleet_df()` (in module `project_fleet`), 25  
`create_fleet_totals_dict()` (`fleet_totals_dict.FleetTotals` method), 18  
`create_new_attributes()` (`fleet_averages_dict.FleetAverages` method), 19  
`create_new_attributes()` (`fleet_totals_dict.FleetTotals` method), 17  
`create_project_dict()` (`project_dicts.InputFileDict` method), 25  
`create_regclass_sales_dict()` (`fleet_totals_dict.FleetTotals` method), 18  
`create_sourcetype_sales_dict()` (`fleet_totals_dict.FleetTotals` method), 18  
`create_weighted_cost_dict()` (in module `weighted_results`), 29  
`CreateFigures` (class in `figures`), 17

## D

`def_costs` module, 13  
`deflator_df()` (`get_context_data.GetDeflators` method), 24

direct\_costs  
     module, 14  
 discount\_values() (*in module discounting*), 16  
 discounting  
     module, 16

## E

emission\_costs  
     module, 16

## F

figures  
     module, 17  
 fleet\_averages\_dict  
     module, 19  
 fleet\_totals\_dict  
     module, 17  
 FleetAverages (*class in fleet\_averages\_dict*), 19  
 FleetTotals (*class in fleet\_totals\_dict*), 17  
 fuel\_costs  
     module, 20  
 fueltype\_name() (*vehicle.Vehicle method*), 28

## G

general\_functions  
     module, 21  
 get\_attribute\_value()  
     (*fleet\_averages\_dict.FleetAverages method*), 19  
 get\_attribute\_value() (*fleet\_totals\_dict.FleetTotals method*), 19  
 get\_attribute\_value() (*project\_dicts.InputFileDict method*), 25  
 get\_common\_metrics() (*in module general\_functions*), 22  
 get\_context\_data  
     module, 23  
 get\_criteria\_cost\_factors() (*in module emission\_costs*), 16  
 get\_file\_datetime() (*in module general\_functions*), 22  
 get\_orvr\_adjustment() (*in module fuel\_costs*), 20  
 get\_prices() (*get\_context\_data.GetFuelPrices method*), 23  
 GetDeflators (*class in get\_context\_data*), 24  
 GetFuelPrices (*class in get\_context\_data*), 23

## I

indirect\_costs  
     module, 24  
 InputFileDict (*class in project\_dicts*), 25  
 inputs\_filenames() (*in module general\_functions*), 21

## L

line\_chart\_arg\_by\_options() (*figures.CreateFigures method*), 17  
 line\_chart\_args\_by\_option() (*figures.CreateFigures method*), 17

## M

main() (*in module tool\_main*), 28  
 melt\_df() (*get\_context\_data.GetDeflators method*), 24  
 melt\_df() (*get\_context\_data.GetFuelPrices method*), 23  
 module  
     calc\_deltas, 13  
     def\_costs, 13  
     direct\_costs, 14  
     discounting, 16  
     emission\_costs, 16  
     figures, 17  
     fleet\_averages\_dict, 19  
     fleet\_totals\_dict, 17  
     fuel\_costs, 20  
     general\_functions, 21  
     get\_context\_data, 23  
     indirect\_costs, 24  
     project\_dicts, 25  
     project\_fleet, 25  
     repair\_costs, 26  
     sum\_by\_vehicle, 27  
     tech\_costs, 28  
     tool\_main, 28  
     vehicle, 28  
     weighted\_results, 29

## O

option\_name() (*vehicle.Vehicle static method*), 29

## P

project\_dicts  
     module, 25  
 project\_fleet  
     module, 25

## R

read\_input\_files() (*in module general\_functions*), 22  
 regclass\_name() (*vehicle.Vehicle method*), 28  
 regclass\_vehicles() (*in module project\_fleet*), 26  
 repair\_costs  
     module, 26  
 reshape\_df() (*in module general\_functions*), 21  
 round\_metrics() (*in module general\_functions*), 22  
 round\_sig() (*in module general\_functions*), 22  
 row\_dict() (*get\_context\_data.GetFuelPrices method*), 23

## S

[save\\_dict\\_to\\_csv\(\)](#) (in module *general\_functions*), 22  
[select\\_aeo\\_table\\_rows\(\)](#)  
     (*get\_context\_data.GetFuelPrices* method),  
     23  
[sourcetype\\_name\(\)](#) (*vehicle.Vehicle* method), 28  
[sourcetype\\_vehicles\(\)](#) (in module *project\_fleet*), 26  
[sum\\_by\\_vehicle](#)  
     module, 27

## T

[tech\\_costs](#)  
     module, 28  
[tech\\_package\\_cost\(\)](#) (in module *direct\_costs*), 14  
[tech\\_pkg\\_cost\\_withlearning\(\)](#) (in module *direct\_costs*), 15  
[tool\\_main](#)  
     module, 28

## U

[update\\_dict\(\)](#) (*fleet\_averages\_dict.FleetAverages*  
     *method*), 19  
[update\\_dict\(\)](#) (*fleet\_totals\_dict.FleetTotals* method),  
     19

## V

[vehicle](#)  
     module, 28  
[Vehicle](#) (class in *vehicle*), 28  
[vehicle\\_name\(\)](#) (*vehicle.Vehicle* static method), 28

## W

[weighted\\_results](#)  
     module, 29