# Project 1: Armageddon - The hazard of small asteroids

## Synopsis:

Asteroids entering Earth's atmosphere are subject to extreme drag forces that decelerate, heat and disrupt the space rocks. The fate of an asteroid is a complex function of its initial mass, speed, trajectory angle and internal strength.

Asteroids 10-100 m in diameter can penetrate deep into Earth's atmosphere and disrupt catastrophically, generating an atmospheric disturbance (airburst) that can cause damage on the ground. Such an event occurred over the city of Chelyabinsk in Russia, in 2013, releasing energy equivalent to about 520 kilotons of TNT (1 kt TNT is equivalent to $4.184 \times 10^{12}$ J), and injuring thousands of people (Popova et al., 2013; Brown et al., 2013). An even larger event occurred over Tunguska, a relatively unpopulated area in Siberia, in 1908.

This simulator predicts the fate of asteroids entering Earth's atmosphere, and provides a hazard mapper for an impact over the UK.

## Problem definition

### Equations of motion for a rigid asteroid

The dynamics of an asteroid in Earth's atmosphere prior to break-up is governed by a coupled set of ordinary differential equations:

$$\begin{aligned}
\frac{dv}{dt} &= \frac{-C_D \rho_a A v^2}{2m} + g \sin\theta \\
\frac{dm}{dt} &= \frac{-C_H \rho_a A v^3}{2Q} \\
\frac{d\theta}{dt} &= \frac{g\cos\theta}{v} - \frac{C_L \rho_a A v}{2m} - \frac{v\cos\theta}{R_P + z} \\
\frac{dz}{dt} &= -v\sin\theta \\
\frac{dx}{dt} &= \frac{v\cos\theta}{1 + z/R_P}
\end{aligned}$$

In these equations, $v$, $m$, and $A$ are the asteroid speed (along trajectory), mass and cross-sectional area, respectively. We will assume an initially **spherical asteroid** to convert from inital radius to mass (and cross-sectional area). $\theta$ is the meteoroid trajectory angle to the horizontal (in radians), $x$ is the downrange distance of the meteoroid from its entry position, $z$ is the altitude and $t$ is time; $C_D$ is the drag coefficient, $\rho_a$ is the atmospheric density (a function of altitude ), $C_H$ is an ablation efficiency coefficient, $Q$ is the specific heat of ablation; $C_L$ is a lift coefficient; and $R_P$ is the planetary radius. All terms use MKS units.

### Asteroid break-up and deformation

A commonly used criterion for the break-up of an asteroid in the atmosphere is when the ram pressure of the air interacting with the asteroid $\rho_a v^2$ first exceeds the strength of the asteroid $Y$.

$$\rho_a v^2 = Y$$

Should break-up occur, the asteroid deforms and spreads laterally as it continues its passage through the atmosphere. Several models for the spreading rate have been proposed. In the simplest model, the fragmented asteroid's spreading rate is related to its along trajectory speed (Hills and Goda, 1993):

$$\frac{dr}{dt} = \left[\frac{7}{2}\alpha\frac{\rho_a}{\rho_m}\right]^{1/2} v$$

Where $r$ is the asteroid radius, $\rho_m$ is the asteroid density (assumed constant) and $\alpha$ is a spreading coefficient, often taken to be 0.3. It is conventional to define the cross-sectional area of the expanding cloud of fragments as $A = \pi r^2$ (i.e., assuming a circular cross-section), for use in the above equations. Fragmentation and spreading **ceases** when the ram pressure drops back below the strength of the meteoroid $\rho_a v^2 < Y$.

## Airblast damage

The rapid deposition of energy in the atmosphere is analogous to an explosion and so the environmental consequences of the airburst can be estimated using empirical data from atmospheric explosion experiments (Glasstone and Dolan, 1977).

The main cause of damage close to the impact site is a strong (pressure) blastwave in the air, known as the **airblast**. Empirical data suggest that the pressure in this wave $p$ (in Pa) (above ambient, also known as overpressure), as a function of explosion energy $E_k$ (in kilotons of TNT equivalent), burst altitude $z_b$ (in m) and horizontal range $r$ (in m), is given by:

$$p(r) = 3.14 \times 10^{11} \left( \frac{r^2 + z_b^2}{E_k^{2/3}} \right)^{-1.3} + 1.8 \times 10^{7} \left( \frac{r^2 + z_b^2}{E_k^{2/3}} \right)^{-0.565}$$

For airbursts, we will take the total kinetic energy lost by the asteroid at the burst altitude as the burst energy $E_k$. For low-altitude airbursts or cratering events, we will define $E_k$ as the **larger** of the total kinetic energy lost by the asteroid at the burst altitude or the residual kinetic energy of the asteroid when it hits the ground.

The following threshold pressures can then be used to define different degrees of damage.

| Damage Level | Description | Pressure (kPa) |
|---|---|---|
| 1 | ~10% glass windows shatter | 1.0 |
| 2 | ~90% glass windows shatter | 3.5 |
| 3 | Wood frame buildings collapse | 27 |
| 4 | Multistory brick buildings collapse | 43 |

Table 1: Pressure thresholds (in kPa) for airblast damage

## Additional sections

You should expand this documentation to include explanatory text for all components of your tool.

# Function API

Python asteroid airburst calculator

*class* `armageddon.Planet`(*atmos_func='exponential'*, *atmos_filename=None*, *Cd=1.0*, *Ch=0.1*, *Q=10000000.0*, *Cl=0.001*, *alpha=0.3*, *Rp=6371000.0*, *g=9.81*, *H=8000.0*, *rho0=1.2*)

The class called Planet is initialised with constants appropriate for the given target planet, including the atmospheric density profile and other constants

Set up the initial parameters and constants for the target planet

**Parameters:**
- **atmos_func** (*string, optional*) – Function which computes atmospheric density, rho, at altitude, z. Default is the exponential function rho = rho0 exp(-z/H). Options are ‘exponential’, ‘tabular’ and ‘constant’

- **atmos_filename** (*string, optional*) – Name of the filename to use with the tabular atmos_func option
- **Cd** (*float, optional*) – The drag coefficient
- **Ch** (*float, optional*) – The heat transfer coefficient
- **Q** (*float, optional*) – The heat of ablation (J/kg)
- **Cl** (*float, optional*) – Lift coefficient
- **alpha** (*float, optional*) – Dispersion coefficient
- **Rp** (*float, optional*) – Planet radius (m)
- **rhoo** (*float, optional*) – Air density at zero altitude (kg/m^3)
- **g** (*float, optional*) – Surface gravity (m/s^2)
- **H** (*float, optional*) – Atmospheric scale height (m)

`RK4`(*f*, *y0*, *t0*, *t_max*, *dt*, *strength*, *density*)

Solves ODE using explicit 4-stage, 4th order Runge-Kutta method

**Parameters:**
- **f** (*function*) – Returns derivative
- **y0** (*np.array*) – '1 x j' array Initial vector of j parameters. In this script j=6, namely velocity, mass, angle, altitude, distance, and radius
- **t0** (*float*) – Initial time for time-stepping
- **t_max** (*float*) – Final time for time-stepping
- **dt** (*float*) – Time interval for time-stepping
- **strength** (*float*) – The strength of the asteroid
- **density** (*float*) – The density of the asteroid

**Returns:**
- **y** (*np.array*) – 'n x j' array of solution over full time frame
- **t** (*np.array*) – '1 x n' array of times matching array y, up to t_max

`RK45`(*f*, *y0*, *t0*, *t_max*, *dt*, *strength*, *density*, *tol=1e-07*, *error_out=False*)

Solves ODE using explicit 7-stage, 5th order Runge-Kutta method, aka Dormand-Prince method, with adaptive time-stepping :param f: Returns derivative :type f: function :param y0: '1 x 6' array

Initial vector of velocity, mass, angle, altitude, distance, and radius

**Parameters:**
- **t0** (*float*) – Initial time for time-stepping
- **t_max** (*float*) – Final time for time-stepping
- **dt** (*float*) – Time interval for time-stepping
- **strength** (*float*) – The strength of the asteroid
- **density** (*float*) – The density of the asteroid
- **tol** (*float, optional*) – Tolerance for error of solution
- **error_out** (*bool, optional*) – Whether error wants to be part of the output

**Returns:**
- **y_out** (*np.array*) – 'n x 6' array of velocities, masses, angles, altitudes, distances, and radii over full time frame
- **t_out** (*float*) – '1 x n' array of times through time-stepping, up to t_max
- **e_out** (*array, conditional*) – '1 x n' array of corresponding errors. Only part of output if error_out is True

`analyse_outcome`(*result*)

Inspect a pre-found solution to calculate the impact and airburst stats

**Parameters:** **result** (*DataFrame*) – pandas dataframe with velocity, mass, angle, altitude, horizontal distance, radius and dedz as a function of time

**Returns:** **outcome** – dictionary with details of the impact event, which should contain the key `outcome` (which should contain one of the following strings: `Airburst`, `Cratering` or `Airburst and cratering`), as well as the following keys: `burst_peak_dedz`, `burst_altitude`, `burst_distance`, `burst_energy`

**Return type:** Dict

`calculate_energy`(*result*)

Function to calculate the kinetic energy lost per unit altitude in kilotons TNT per km, for a given solution.

**Parameters:**

- **result** (*DataFrame*) – A pandas dataframe with columns for the velocity, mass, angle, altitude, horizontal distance and radius as a function of time
- **Returns** (*DataFrame*) – Returns the dataframe with additional column `dedz` which is the kinetic energy lost per unit altitude

**Returns:** **result**

**Return type:** DataFrame

`determine_parameters`(*rho0=3300*, *theta0=18.3*, *v0=19200.0*, *radians=False*)

Determine the strength (Y) and radius (r), given initial data rho0, theta0, v0, and observed energy deposition curve.

**Parameters:**
- **rho0** (*float, optional*) – The density.
- **theta0** (*float, optional*) – The impact angle.
- **v0** (*float, optional,*) – The entry velocity
- **radians** (*bool, optional*) – Whether theta0 is in radians or not

**Returns:**
- **Y** (*float*) – Predicted strength
- **r** (*float*) – Predicted radius

`error_convergence`(*radius*, *velocity*, *density*, *strength*, *angle*, *tol_list*, *init_altitude=100000.0*, *dt=0.05*, *radians=False*)

Calculates errors depending on the tolerance limit

**Parameters:** **tol_list** (*array*) – '1 x n' array of n tolerances to test for solver

**Returns:** **err_list** – '1 x n' array of n errors corresponding to n tolerances

**Return type:** array

`f`(*t*, *y*, *strength*, *density*)

Calculates derivative of input vector y, as given in equations of motion found in AirburstSolver.ipynb

**Parameters:**
- **t** (*float*) – Current time
- **y** (*np.array*) – '1 x 6' array containing current velocity, mass, angle, altitude, distance, and radius
- **strength** (*float*) – Strength of the asteroid
- **density** (*float*) – Density of the asteroid

**Returns:** **out** – '1 x 6' array of current derivatives of velocity, mass, angle, altitude, distance, and radius

**Return type:** np.array

`solve_atmospheric_entry`(*radius*, *velocity*, *density*, *strength*, *angle*, *init_altitude=100000.0*, *dt=0.05*, *radians=False*)

Solve the system of differential equations for a given impact scenario

**Parameters:**
- **radius** (*float*) – The radius of the asteroid in meters
- **velocity** (*float*) – The entery speed of the asteroid in meters/second
- **density** (*float*) – The density of the asteroid in kg/m^3
- **strength** (*float*) – The strength of the asteroid (i.e. the maximum pressure it can take before fragmenting) in N/m^2
- **angle** (*float*) – The initial trajectory angle of the asteroid to the horizontal By default, input is in degrees. If 'radians' is set to True, the input should be in radians
- **init_altitude** (*float, optional*) – Initial altitude in m
- **dt** (*float, optional*) – The output timestep, in s
- **radians** (*logical, optional*) – Whether angles should be given in degrees or radians. Default=False Angles returned in the dataframe will have the same units as the input

**Returns:** **Result** – A pandas dataframe containing the solution to the system. Includes the following columns: 'velocity', 'mass', 'angle', 'altitude', 'distance', 'radius', 'time'

**Return type:** DataFrame

*class* `armageddon.PostcodeLocator`(*postcode_file='./resources/full_postcodes.csv'*, *census_file='./resources/population_by_postcode_sector.csv'*, *norm=<function great_circle_distance>*)

Class to interact with a postcode database file.

**Parameters:**
- **postcode_file** (*str, optional*) – Filename of a .csv file containing geographic location data for postcodes.
- **census_file** (*str, optional*) – Filename of a .csv file containing census data by postcode sector.
- **norm** (*function*) – Python function defining the distance between points in latitude-longitude space.

`get_population_of_postcode`(*postcodes*, *sector=False*)

Return populations of a list of postcode units or sectors.

**Parameters:**
- **postcodes** (*list of lists*) – list of postcode units or postcode sectors
- **sector** (*bool, optional*) – if true return populations for postcode sectors, otherwise postcode units

**Returns:** Contains the populations of input postcode units or sectors

**Return type:** list of lists

**Examples**

```
>>> locator = PostcodeLocator()
>>> locator.get_population_of_postcode([['SW7 2AZ','SW7 2BT','SW7 2BU','SW7 2DD']])
[[18.71311475409836, 18.71311475409836, 18.71311475409836, 18.71311475409836]]
>>> locator.get_population_of_postcode([['SW7  2']], True)
[[2283]]
```

`get_postcodes_by_radius`(*X*, *radii*, *sector=False*)

Return (unit or sector) postcodes within specific distances of input location.

**Parameters:**
- **X** (*arraylike*) – Latitude-longitude pair of centre location
- **radii** (*arraylike*) – array of radial distances from X
- **sector** (*bool, optional*) – if true return postcode sectors, otherwise postcode units

**Returns:** Contains the lists of postcodes closer than the elements of radii to the location X.

**Return type:** list of lists

**Examples**

```
>>> locator = PostcodeLocator()
>>> locator.get_postcodes_by_radius((51.4981, -0.1773), [0.13e3])
[['SW7 2AZ', 'SW7 2BT', 'SW7 2BU', 'SW7 2DD', 'SW7 5HF', 'SW7 5HG', 'SW7 5HQ']]
>>> locator.get_postcodes_by_radius((51.4981, -0.1773), [0.4e3, 0.2e3], True)
[['SW7 1', 'SW7 4', 'SW7 3', 'SW7 2', 'SW7 9', 'SW7 5'], ['SW7 1', 'SW7 4', 'SW7 3',
```

`armageddon.damage_zones`(*outcome*, *lat*, *lon*, *bearing*, *pressures*)

Calculate the latitude and longitude of the surface zero location and the list of airblast damage radii (m) for a given impact scenario.

**Parameters:**
- **outcome** (*Dict*) – the outcome dictionary from an impact scenario
- **lat** (*float*) – latitude of the meteoroid entry point (degrees)
- **lon** (*float*) – longitude of the meteoroid entry point (degrees)
- **bearing** (*float*) – Bearing (azimuth) relative to north of meteoroid trajectory (degrees)
- **pressures** (*float, arraylike*) – List of threshold pressures to define airblast damage levels

**Returns:**
- **blat** (*float*) – latitude of the surface zero point (degrees)
- **blon** (*float*) – longitude of the surface zero point (degrees)
- **damrad** (*arraylike, float*) – List of distances specifying the blast radii for the input damage levels

**Examples**

```
>>> import armageddon
>>> outcome = {'burst_altitude': 8e3, 'burst_energy': 7e3,
               'burst_distance': 90e3, 'burst_peak_dedz': 1e3,
               'outcome': 'Airburst'}
>>> armageddon.damage_zones(outcome, 52.79, -2.95, 135, pressures=[1e3, 3.5e3, 27e3, 43e
```

armageddon.**great_circle_distance**(*latlon1*, *latlon2*)

Calculate the great circle distance (in metres) between pairs of points specified as latitude and longitude on a spherical Earth (with radius 6371 km).

**Parameters:**
- **latlon1** (*arraylike*) – latitudes and longitudes of first point (as [n, 2] array for n points)
- **latlon2** (*arraylike*) – latitudes and longitudes of second point (as [m, 2] array for m points)

**Returns:** Distance in metres between each pair of points (as an n x m array)

**Return type:** numpy.ndarray

**Examples**

```
>>> import numpy
>>> fmt = lambda x: numpy.format_float_scientific(x, precision=3)
>>> with numpy.printoptions(formatter={'all', fmt}): print(great_circle_distance([[54.0,
'[1.286e+05 6.378e+04]'
```

armageddon.**impact_risk**(*planet*, *means={'angle': 20, 'bearing': - 45.0, 'density': 3000, 'lat': 51.5, 'lon': 1.5, 'radius': 10, 'strength': 1000000.0, 'velocity': 19000.0}*, *stdevs={'angle': 1, 'bearing': 0.5, 'density': 500, 'lat': 0.025, 'lon': 0.025, 'radius': 1, 'strength': 500000.0, 'velocity': 1000.0}*, *pressure=27000.0*, *nsamples=100*, *sector=True*)

Perform an uncertainty analysis to calculate the risk for each affected UK postcode or postcode sector

**Parameters:**
- **planet** (*armageddon.Planet instance*) – The Planet instance from which to solve the atmospheric entry
- **means** (*dict*) – A dictionary of mean input values for the uncertainty analysis. This should include values for `radius`, `angle`, `strength`, `density`, `velocity`, `lat`, `lon` and `bearing`
- **stdevs** (*dict*) – A dictionary of standard deviations for each input value. This should include values for `radius`, `angle`, `strength`, `density`, `velocity`, `lat`, `lon` and `bearing`
- **pressure** (*float*) – The pressure at which to calculate the damage zone for each impact
- **nsamples** (*int*) – The number of iterations to perform in the uncertainty analysis
- **sector** (*logical, optional*) – If True (default) calculate the risk for postcode sectors, otherwise calculate the risk for postcodes

**Returns:** **risk** – A pandas DataFrame with columns for postcode (or postcode sector) and the associated risk. These should be called `postcode` or `sector`, and `risk`.

**Return type:** DataFrame

armageddon.**minimize**(*fun*, *x0*, *args=()*, *method=None*, *jac=None*, *hess=None*, *hessp=None*, *bounds=None*, *constraints=()*, *tol=None*, *callback=None*, *options=None*)

Minimization of scalar function of one or more variables.

**Parameters:**
- **fun** (*callable*) –
  The objective function to be minimized.
  ```
  fun(x, *args) -> float
  ```
  where `x` is an 1-D array with shape (n,) and `args` is a tuple of the fixed parameters needed to completely specify the function.
- **x0** (*ndarray, shape (n,)*) – Initial guess. Array of real elements of size (n,), where 'n' is the number of independent variables.

- **args** (*tuple, optional*) – Extra arguments passed to the objective function and its derivatives (*fun*, *jac* and *hess* functions).
- **method** (*str or callable, optional*) –
  Type of solver. Should be one of
  - 'Nelder-Mead' (see here)
  - 'Powell' (see here)
  - 'CG' (see here)
  - 'BFGS' (see here)
  - 'Newton-CG' (see here)
  - 'L-BFGS-B' (see here)
  - 'TNC' (see here)
  - 'COBYLA' (see here)
  - 'SLSQP' (see here)
  - 'trust-constr'(see here)
  - 'dogleg' (see here)
  - 'trust-ncg' (see here)
  - 'trust-exact' (see here)
  - 'trust-krylov' (see here)
  - custom - a callable object (added in version 0.14.0), see below for description.

  If not given, chosen to be one of `BFGS`, `L-BFGS-B`, `SLSQP`, depending if the problem has constraints or bounds.
- **jac** (*{callable, '2-point', '3-point', 'cs', bool}, optional*) –
  Method for computing the gradient vector. Only for CG, BFGS, Newton-CG, L-BFGS-B, TNC, SLSQP, dogleg, trust-ncg, trust-krylov, trust-exact and trust-constr. If it is a callable, it should be a function that returns the gradient vector:
  ```
  jac(x, *args) -> array_like, shape (n,)
  ```
  where `x` is an array with shape (n,) and `args` is a tuple with the fixed parameters. If *jac* is a Boolean and is True, *fun* is assumed to return and objective and gradient as an `(f, g)` tuple. Methods 'Newton-CG', 'trust-ncg', 'dogleg', 'trust-exact', and 'trust-krylov' require that either a callable be supplied, or that *fun* return the objective and gradient. If None or False, the gradient will be estimated using 2-point finite difference estimation with an absolute step size. Alternatively, the keywords {'2-point', '3-point', 'cs'} can be used to select a finite difference scheme for numerical estimation of the gradient with a relative step size. These finite difference schemes obey any specified *bounds*.
- **hess** (*{callable, '2-point', '3-point', 'cs', HessianUpdateStrategy}, optional*) –
  Method for computing the Hessian matrix. Only for Newton-CG, dogleg, trust-ncg, trust-krylov, trust-exact and trust-constr. If it is callable, it should return the Hessian matrix:
  ```
  hess(x, *args) -> {LinearOperator, spmatrix, array}, (n, n)
  ```
  where x is a (n,) ndarray and *args* is a tuple with the fixed parameters. LinearOperator and sparse matrix returns are allowed only for 'trust-constr' method. Alternatively, the keywords {'2-point', '3-point', 'cs'} select a finite difference scheme for numerical estimation. Or, objects implementing *HessianUpdateStrategy* interface can be used to approximate the Hessian. Available quasi-Newton methods implementing this interface are:
  - *BFGS*;
  - *SR1*.

  Whenever the gradient is estimated via finite-differences, the Hessian cannot be estimated with options {'2-point', '3-point', 'cs'} and needs to be estimated using one of the quasi-Newton strategies. Finite-difference options {'2-point', '3-point', 'cs'} and *HessianUpdateStrategy* are available only for 'trust-constr' method.
- **hessp** (*callable, optional*) –
  Hessian of objective function times an arbitrary vector p. Only for Newton-CG, trust-ncg, trust-krylov, trust-constr. Only one of *hessp* or *hess* needs to be given. If *hess* is provided, then *hessp* will be ignored. *hessp* must compute the Hessian times an arbitrary vector:
  ```
  hessp(x, p, *args) ->  ndarray shape (n,)
  ```

where x is a (n,) ndarray, p is an arbitrary vector with dimension (n,) and *args* is a tuple with the fixed parameters.

- **bounds** (sequence or *Bounds*, optional) –
  Bounds on variables for L-BFGS-B, TNC, SLSQP, Powell, and trust-constr methods. There are two ways to specify the bounds:
  1. Instance of *Bounds* class.
  2. Sequence of `(min, max)` pairs for each element in *x*. None is used to specify no bound.
- **constraints** (*{Constraint, dict} or List of {Constraint, dict}, optional*) –
  Constraints definition (only for COBYLA, SLSQP and trust-constr).
  Constraints for 'trust-constr' are defined as a single object or a list of objects specifying constraints to the optimization problem. Available constraints are:
  - *LinearConstraint*
  - *NonlinearConstraint*

  Constraints for COBYLA, SLSQP are defined as a list of dictionaries. Each dictionary with fields:

  type : *str*
  Constraint type: 'eq' for equality, 'ineq' for inequality.

  fun : *callable*
  The function defining the constraint.

  jac : *callable, optional*
  The Jacobian of *fun* (only for SLSQP).

  args : *sequence, optional*
  Extra arguments to be passed to the function and Jacobian.

  Equality constraint means that the constraint function result is to be zero whereas inequality means that it is to be non-negative. Note that COBYLA only supports inequality constraints.
- **tol** (*float, optional*) – Tolerance for termination. For detailed control, use solver-specific options.
- **options** (*dict, optional*) –
  A dictionary of solver options. All methods accept the following generic options:

  maxiter : *int*
  Maximum number of iterations to perform. Depending on the method each iteration may use several function evaluations.

  disp : *bool*
  Set to True to print convergence messages.

  For method-specific options, see **`show_options()`**.
- **callback** (*callable, optional*) –
  Called after each iteration. For 'trust-constr' it is a callable with the signature:
  ```
  callback(xk, OptimizeResult state) -> bool
  ```
  where `xk` is the current parameter vector. and `state` is an *OptimizeResult* object, with the same fields as the ones from the return. If callback returns True the algorithm execution is terminated. For all the other methods, the signature is:
  ```
  callback(xk)
  ```
  where `xk` is the current parameter vector.

**Returns:** **res** – The optimization result represented as a `OptimizeResult` object. Important attributes are: `x` the solution array, `success` a Boolean flag indicating if the optimizer exited successfully and `message` which describes the cause of the termination. See *OptimizeResult* for a description of other attributes.

**Return type:** OptimizeResult

## See also:

**`minimize_scalar()`**
Interface to minimization algorithms for scalar univariate functions

`show_options()`
: Additional options accepted by the solvers

### Notes

This section describes the available solvers that can be selected by the 'method' parameter. The default method is *BFGS*.

### Unconstrained minimization

Method Nelder-Mead uses the Simplex algorithm [1], [2]. This algorithm is robust in many applications. However, if numerical computation of derivative can be trusted, other algorithms using the first and/or second derivatives information might be preferred for their better performance in general.

Method CG uses a nonlinear conjugate gradient algorithm by Polak and Ribiere, a variant of the Fletcher-Reeves method described in [5] pp.120-122. Only the first derivatives are used.

Method BFGS uses the quasi-Newton method of Broyden, Fletcher, Goldfarb, and Shanno (BFGS) [5] pp. 136. It uses the first derivatives only. BFGS has proven good performance even for non-smooth optimizations. This method also returns an approximation of the Hessian inverse, stored as *hess_inv* in the OptimizeResult object.

Method Newton-CG uses a Newton-CG algorithm [5] pp. 168 (also known as the truncated Newton method). It uses a CG method to the compute the search direction. See also *TNC* method for a box-constrained minimization with a similar algorithm. Suitable for large-scale problems.

Method dogleg uses the dog-leg trust-region algorithm [5] for unconstrained minimization. This algorithm requires the gradient and Hessian; furthermore the Hessian is required to be positive definite.

Method trust-ncg uses the Newton conjugate gradient trust-region algorithm [5] for unconstrained minimization. This algorithm requires the gradient and either the Hessian or a function that computes the product of the Hessian with a given vector. Suitable for large-scale problems.

Method trust-krylov uses the Newton GLTR trust-region algorithm [14], [15] for unconstrained minimization. This algorithm requires the gradient and either the Hessian or a function that computes the product of the Hessian with a given vector. Suitable for large-scale problems. On indefinite problems it requires usually less iterations than the *trust-ncg* method and is recommended for medium and large-scale problems.

Method trust-exact is a trust-region method for unconstrained minimization in which quadratic subproblems are solved almost exactly [13]. This algorithm requires the gradient and the Hessian (which is *not* required to be positive definite). It is, in many situations, the Newton method to converge in fewer iteraction and the most recommended for small and medium-size problems.

### Bound-Constrained minimization

Method L-BFGS-B uses the L-BFGS-B algorithm [6], [7] for bound constrained minimization.

Method Powell is a modification of Powell's method [3], [4] which is a conjugate direction method. It performs sequential one-dimensional minimizations along each vector of the directions set (*direc* field in *options* and *info*), which is updated at each iteration of the main minimization loop. The function need not be differentiable, and no derivatives are taken. If bounds are not provided, then an unbounded line search will be used. If bounds are provided and the initial guess is within the bounds, then every function evaluation throughout the minimization procedure will be within the bounds. If bounds are provided, the initial guess is outside the bounds, and *direc* is full rank (default has full rank), then some function evaluations during the first iteration may be outside the bounds, but every function evaluation after the first iteration will be within the bounds. If *direc* is not full rank, then some parameters may not be optimized and the solution is not guaranteed to be within the bounds.

Method TNC uses a truncated Newton algorithm [5], [8] to minimize a function with variables subject to bounds. This algorithm uses gradient information; it is also called Newton Conjugate-Gradient. It differs from the *Newton-CG* method described above as it wraps a C implementation and allows each variable to be given upper and lower bounds.

**Constrained Minimization**

Method COBYLA uses the Constrained Optimization BY Linear Approximation (COBYLA) method [9], [10], [11]. The algorithm is based on linear approximations to the objective function and each constraint. The method wraps a FORTRAN implementation of the algorithm. The constraints functions 'fun' may return either a single number or an array or list of numbers.

Method SLSQP uses Sequential Least SQuares Programming to minimize a function of several variables with any combination of bounds, equality and inequality constraints. The method wraps the SLSQP Optimization subroutine originally implemented by Dieter Kraft [12]. Note that the wrapper handles infinite values in bounds by converting them into large floating values.

Method trust-constr is a trust-region algorithm for constrained optimization. It swiches between two implementations depending on the problem definition. It is the most versatile constrained minimization algorithm implemented in SciPy and the most appropriate for large-scale problems. For equality constrained problems it is an implementation of Byrd-Omojokun Trust-Region SQP method described in [17] and in [5], p. 549. When inequality constraints are imposed as well, it swiches to the trust-region interior point method described in [16]. This interior point algorithm, in turn, solves inequality constraints by introducing slack variables and solving a sequence of equality-constrained barrier problems for progressively smaller values of the barrier parameter. The previously described equality constrained SQP method is used to solve the subproblems with increasing levels of accuracy as the iterate gets closer to a solution.

**Finite-Difference Options**

For Method trust-constr the gradient and the Hessian may be approximated using three finite-difference schemes: {'2-point', '3-point', 'cs'}. The scheme 'cs' is, potentially, the most accurate but it requires the function to correctly handles complex inputs and to be differentiable in the complex plane. The scheme '3-point' is more accurate than '2-point' but requires twice as many operations.

**Custom minimizers**

It may be useful to pass a custom minimization method, for example when using a frontend to this method such as *scipy.optimize.basinhopping* or a different library. You can simply pass a callable as the `method` parameter.

The callable is called as `method(fun, x0, args, **kwargs, **options)` where `kwargs` corresponds to any other parameters passed to *minimize* (such as *callback*, *hess*, etc.), except the *options* dict, which has its contents also passed as *method* parameters pair by pair. Also, if *jac* has been passed as a bool type, *jac* and *fun* are mangled so that *fun* returns just the function values and *jac* is converted to a function returning the Jacobian. The method shall return an *OptimizeResult* object.

The provided *method* callable must be able to accept (and possibly ignore) arbitrary parameters; the set of parameters accepted by *minimize* may expand in future versions and then these parameters will be passed to the method. You can find an example in the scipy.optimize tutorial.

*New in version 0.11.0.*

**References**


[1] Nelder, J A, and R Mead. 1965. A Simplex Method for Function Minimization. The Computer Journal 7: 308-13.

[2] Wright M H. 1996. Direct search methods: Once scorned, now respectable, in Numerical Analysis 1995: Proceedings of the 1995 Dundee Biennial Conference in Numerical Analysis (Eds. D F Griffiths and G A Watson). Addison Wesley Longman, Harlow, UK. 191-208.

[3] Powell, M J D. 1964. An efficient method for finding the minimum of a function of several variables without calculating derivatives. The Computer Journal 7: 155-162.

[4] Press W, S A Teukolsky, W T Vetterling and B P Flannery. Numerical Recipes (any edition), Cambridge University Press.

5(1,2,3,4,5,6,7,8) Nocedal, J, and S J Wright. 2006. Numerical Optimization. Springer New York.

[6] Byrd, R H and P Lu and J. Nocedal. 1995. A Limited Memory Algorithm for Bound Constrained Optimization. SIAM Journal on Scientific and Statistical Computing 16 (5): 1190-1208.

[7] Zhu, C and R H Byrd and J Nocedal. 1997. L-BFGS-B: Algorithm 778: L-BFGS-B, FORTRAN routines for large scale bound constrained optimization. ACM Transactions on Mathematical Software 23 (4): 550-560.

[8] Nash, S G. Newton-Type Minimization Via the Lanczos Method. 1984. SIAM Journal of Numerical Analysis 21: 770-778.

[9] Powell, M J D. A direct search optimization method that models the objective and constraint functions by linear interpolation. 1994. Advances in Optimization and Numerical Analysis, eds. S. Gomez and J-P Hennart, Kluwer Academic (Dordrecht), 51-67.

[10] Powell M J D. Direct search algorithms for optimization calculations. 1998. Acta Numerica 7: 287-336.

[11] Powell M J D. A view of algorithms for optimization without derivatives. 2007.Cambridge University Technical Report DAMTP 2007/NA03

[12] Kraft, D. A software package for sequential quadratic programming. 1988. Tech. Rep. DFVLR-FB 88-28, DLR German Aerospace Center – Institute for Flight Mechanics, Koln, Germany.

[13] Conn, A. R., Gould, N. I., and Toint, P. L. Trust region methods. 2000. Siam. pp. 169-200.

[14] F. Lenders, C. Kirches, A. Potschka: “trlib: A vector-free implementation of the GLTR method for iterative solution of the trust region problem”, :arxiv:`1611.04718`

[15] N. Gould, S. Lucidi, M. Roma, P. Toint: “Solving the Trust-Region Subproblem using the Lanczos Method”, SIAM J. Optim., 9(2), 504–525, (1999).

[16] Byrd, Richard H., Mary E. Hribar, and Jorge Nocedal. 1999. An interior point algorithm for large-scale nonlinear programming. SIAM Journal on Optimization 9.4: 877-900.

[17] Lalee, Marucha, Jorge Nocedal, and Todd Plantega. 1998. On the implementation of an algorithm for large-scale equality constrained optimization. SIAM Journal on Optimization 8.3: 682-706.

**Examples**

Let us consider the problem of minimizing the Rosenbrock function. This function (and its respective derivatives) is implemented in *rosen* (resp. *rosen_der*, *rosen_hess*) in the *scipy.optimize*.

```
>>> from scipy.optimize import minimize, rosen, rosen_der
```

A simple application of the *Nelder-Mead* method is:

```
>>> x0 = [1.3, 0.7, 0.8, 1.9, 1.2]
>>> res = minimize(rosen, x0, method='Nelder-Mead', tol=1e-6)
>>> res.x
array([ 1.,  1.,  1.,  1.,  1.])
```

Now using the *BFGS* algorithm, using the first derivative and a few options:

```
>>> res = minimize(rosen, x0, method='BFGS', jac=rosen_der,
...                options={'gtol': 1e-6, 'disp': True})
Optimization terminated successfully.
```

```
         Current function value: 0.000000
         Iterations: 26
         Function evaluations: 31
         Gradient evaluations: 31
>>> res.x
array([ 1.,  1.,  1.,  1.,  1.])
>>> print(res.message)
Optimization terminated successfully.
>>> res.hess_inv
array([[ 0.00749589,  0.01255155,  0.02396251,  0.04750988,  0.09495377],  # may vary
       [ 0.01255155,  0.02510441,  0.04794055,  0.09502834,  0.18996269],
       [ 0.02396251,  0.04794055,  0.09631614,  0.19092151,  0.38165151],
       [ 0.04750988,  0.09502834,  0.19092151,  0.38341252,  0.7664427 ],
       [ 0.09495377,  0.18996269,  0.38165151,  0.7664427,   1.53713523]])
```

Next, consider a minimization problem with several constraints (namely Example 16.4 from [5]). The objective function is:

```
>>> fun = lambda x: (x[0] - 1)**2 + (x[1] - 2.5)**2
```

There are three constraints defined as:

```
>>> cons = ({'type': 'ineq', 'fun': lambda x:  x[0] - 2 * x[1] + 2},
...         {'type': 'ineq', 'fun': lambda x: -x[0] - 2 * x[1] + 6},
...         {'type': 'ineq', 'fun': lambda x: -x[0] + 2 * x[1] + 2})
```

And variables must be positive, hence the following bounds:

```
>>> bnds = ((0, None), (0, None))
```

The optimization problem is solved using the SLSQP method as:

```
>>> res = minimize(fun, (2, 0), method='SLSQP', bounds=bnds,
...                constraints=cons)
```

It should converge to the theoretical solution (1.4 ,1.7).

`armageddon.`**`plot_circle`**(*lat*, *lon*, *radius*, *map=None*, ***kwargs*)

Plot a circle on a map (creating a new folium map instance if necessary).

**Parameters:**
- **lat** (*float*) – latitude of circle to plot (degrees)
- **lon** (*float*) – longitude of circle to plot (degrees)
- **radius** (*float*) – radius of circle to plot (m)
- **map** (*folium.Map*) – existing map object

**Returns:**

**Return type:** Folium map object

**Examples**

#>>> import folium #>>> armageddon.plot_circle(52.79, -2.95, 1e3, map=None)

`armageddon.`**`plot_polyline`**(*lat*, *lon*, *blat*, *blon*, *map=None*)

Plot a line between the meteoroid entry point and the surface zero location on a map (creating a new folium map instance if necessary).

**Parameters:**
- **blat** (*float*) – latitude of the surface zero point (degrees)
- **blon** (*float*) – longitude of the surface zero point (degrees)
- **lat** (*float*) – latitude of the meteoroid entry point (degrees)
- **lon** (*float*) – longitude of the meteoroid entry point (degrees)
- **map** (*folium.Map*) – existing map object

**Returns:**

**Return type:** Folium map object

`armageddon.`**`randomcolor`**()

Randomly select a color for plotting

**Returns:**

**Return type:** A random color code

`armageddon.`**`second_order_Newton`**(*f*, *dfdx*, *df2dx2*, *initial_guess*, *scaling=1*, *max_iterstep=1000000.0*)

Solving a nolinear equation f(x) = 0 using the 2nd order Newton-Raphson method, with iteration function: phi(x) = x + f(x)*[u(x) + f(x)*w(x)], we could show by theoretical analysis that u(x) = - f(x)/f'(x) and w(x) = - 2f''(x)/ [f'(x)]^3 enables a cubic convergence for this method. but to note choosing the initial guess is tricky for this method.

**Parameters:**
- **f** (*function*) – the non-linear function we need to solve, (eg. f(x) = 0)
- **dfdx** (*function*) – the 1st derivative for function f
- **df2dx2** (*function*) – the 2nd derivative for function f
- **intial_guess** (*float*) – the start point of the iteration
- **(Optional)** (*scaling*) – a scaling factor to ensure convergence, depending on the specific function we have. the point here to make the absolute value of f, dfdx, df2dx2 at the fix point C to be small, (eg. |f(C)|, |f'(C)|, |f''(C| is not too big)
- **max_iterstep** (*int*) – the max iterations that you can accept for this solver, by default it's 10^6.

**Returns:** **root** – the numerical solution for the Newton method

**Return type:** float