SciPY con Python

¿Qué es SciPy y por qué es importante?

- SciPy (Scientific Python) es una biblioteca de Python construida sobre Numpy que proporciona herramientas avanzadas para cálculos científicos, matemáticos de ingeniería
- Es ampliamente utilizada en:
  - Álgebra Lineal
  - Optimización
  - Estadística
  - Procesamiento de Señales
  - Integración y ecuaciones diferenciales.
  - Interpolación
  - y mas.

¿Por qué usar SciPy?

- Amplía la capacidades de NumPy con más herramientas matemáticas.
- Es altamente optimizada y eficiente.
- Se integra con Matplotlib, Pandas y otras librerías de análisis de datos.
- Es utilizada en ciencia de datos, machine learning, simulaciones científicas y más.

Instalación y Configuración

- Para instalar SciPy, simplemente usamos pip
- pip install scipy

Verificamos la instalación.

In [1]:
```
import scipy
import numpy as np
```

In [7]:
```
print(scipy.__version__)
print(scipy.__file__)
```

```
1.15.1
c:\Python\Python311\Lib\site-packages\scipy\__init__.py
```

Relación entre SciPy y NumPy

- SciPy se construye sobre NumPy, lo que significa que depende de NumPy para el manejo de arrays.
  - NumPy: Trabaja con estructuras de datos eficientes como ndarray.
  - SciPy: Agrega más herramientas avanzadas a NumPy.

Ejemplo de integración entre NumPy y SciPy

- Creamos matriz de 2*2 con Numpy y calculamos su inversa con SciPy

In [8]:
```
import numpy as np
from scipy import linalg

mat = np.array([[3,2],[1,4]])
mat_inv = linalg.inv(mat)

print('Matriz original: \n', mat)
print('Matriz inversa: \n', mat_inv)
```

```
Matriz original:
 [[3 2]
 [1 4]]
Matriz inversa:
 [[ 0.4 -0.2]
 [-0.1  0.3]]
```

Importación y Estructura de SciPy

- SciPy tiene diferentes módulos especializados en distintas áreas matemáticas.

Estructura principal de SciPy	
scipy.linalg	Algebra Lineal
scipy.optimize	Optimización
scipy.stats	Estadística
scipy.signal	Procesamiento de Señales
scipy.integrate	Integración Numérica
scipy.interpolate	Interpolación
scipy.ndimage	Procesamiento de Imágenes

Ejemplo de importación de módulos

In [2]:
```
from scipy import optimize, linalg, integrate, stats
```

In [3]:
```
mat_A = np.array([[1,3,5],[2,5,1],[2,3,8]])
mat_A
```

Out[3]:
```
array([[1, 3, 5],
       [2, 5, 1],
       [2, 3, 8]])
```

In [4]:
```
linalg.inv(mat_A)
```

Out[4]:
```
array([[-1.48,  0.36,  0.88],
       [ 0.56,  0.08, -0.36],
       [ 0.16, -0.12,  0.04]])
```

In [6]:
```
mat_A.dot(linalg.inv(mat_A))
```

Out[6]:
```
array([[ 1.00000000e+00, -1.11022302e-16,  4.85722573e-17],
       [ 3.05311332e-16,  1.00000000e+00,  7.63278329e-17],
       [ 2.22044605e-16, -1.11022302e-16,  1.00000000e+00]])
```

In [ ]:
```
array = np.array([[2,1],[3,4]])
print('Determinate de array:', linalg.det(array))
```

```
Determinate de array: 5.0
```

In [10]:
```python
def f(x):
    return x**2 + 3*x + 2

res = optimize.minimize(f, x0=0)
print(f'El mínimo de la función {f} es: {res.x}')
```

```
El mínimo de la función <function f at 0x0000026107B163E0> es: [-1.50000001]
```

$f(x) = x^2 + 4x + 4$

In [14]:
```python
def f(x):
    return x**2 - 4*x + 4

minimo = optimize.minimize(f, x0=0)
print(f'El mínimo de la función {f} es: {minimo.x}')
```

```
El mínimo de la función <function f at 0x0000026107B36980> es: [2.00000002]
```

Explorando la Documentación de SciPy

- Podemos acceder a la documentación de cualquier función directamente desde Python
- También podemos visitar la documentación oficial: https://docs.scipy.org/doc/scipy/

In [ ]:
```python
import scipy
help(scipy.optimize.minimize)
```

```
Help on function minimize in module scipy.optimize._minimize:

minimize(fun, x0, args=(), method=None, jac=None, hess=None, hessp=None, bounds=No
ne, 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 a 1-D array with shape (n,) and ``args``
        is a tuple of the fixed parameters needed to completely
        specify the function.

        Suppose the callable has signature ``f0(x, *my_args, **my_kwargs)``, where
        ``my_args`` and ``my_kwargs`` are required positional and keyword argument
s.
        Rather than passing ``f0`` as the callable, wrap it to accept
        only ``x``; e.g., pass ``fun=lambda x: f0(x, *my_args, **my_kwargs)`` as t
he
        callable, where ``my_args`` (tuple) and ``my_kwargs`` (dict) have been
        gathered before invoking this 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' :ref:`(see here) <optimize.minimize-neldermead>`
        - 'Powell'      :ref:`(see here) <optimize.minimize-powell>`
        - 'CG'          :ref:`(see here) <optimize.minimize-cg>`
        - 'BFGS'        :ref:`(see here) <optimize.minimize-bfgs>`
        - 'Newton-CG'   :ref:`(see here) <optimize.minimize-newtoncg>`
        - 'L-BFGS-B'    :ref:`(see here) <optimize.minimize-lbfgsb>`
        - 'TNC'         :ref:`(see here) <optimize.minimize-tnc>`
        - 'COBYLA'      :ref:`(see here) <optimize.minimize-cobyla>`
        - 'COBYQA'      :ref:`(see here) <optimize.minimize-cobyqa>`
        - 'SLSQP'       :ref:`(see here) <optimize.minimize-slsqp>`
        - 'trust-constr':ref:`(see here) <optimize.minimize-trustconstr>`
        - 'dogleg'      :ref:`(see here) <optimize.minimize-dogleg>`
        - 'trust-ncg'   :ref:`(see here) <optimize.minimize-trustncg>`
        - 'trust-exact' :ref:`(see here) <optimize.minimize-trustexact>`
        - 'trust-krylov' :ref:`(see here) <optimize.minimize-trustkrylov>`
        - custom - a callable object, see below for description.

        If not given, chosen to be one of ``BFGS``, ``L-BFGS-B``, ``SLSQP``,
        depending on whether or not 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 a tuple ``(f, g)`` containing the objective
function and the gradient.
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.
The keywords {'2-point', '3-point', 'cs'} can also be used to select
a finite difference scheme for numerical estimation of the hessian.
Alternatively, objects implementing the `HessianUpdateStrategy`
interface can be used to approximate the Hessian. Available
quasi-Newton methods implementing this interface are:

- `BFGS`
- `SR1`

Not all of the options are available for each of the methods; for
availability refer to the notes.

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 Nelder-Mead, L-BFGS-B, TNC, SLSQP, Powell,
trust-constr, COBYLA, and COBYQA 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, COBYQA, SLSQP and trust-constr.

Constraints for 'trust-constr' and 'cobyqa' 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. When `tol` is specified, the selected
    minimization algorithm sets some relevant solver-specific tolerance(s)
    equal to `tol`. For detailed control, use solver-specific
    options.
options : dict, optional
    A dictionary of solver options. All methods except `TNC` accept the
    following generic options:

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

        For `TNC` use `maxfun` instead of `maxiter`.
    disp : bool
        Set to True to print convergence messages.

    For method-specific options, see :func:`show_options()`.
callback : callable, optional
    A callable called after each iteration.

    All methods except TNC, SLSQP, and COBYLA support a callable with
    the signature::

        callback(intermediate_result: OptimizeResult)

    where ``intermediate_result`` is a keyword parameter containing an
    `OptimizeResult` with attributes ``x`` and ``fun``, the present values
    of the parameter vector and objective function. Note that the name
    of the parameter must be ``intermediate_result`` for the callback
    to be passed an `OptimizeResult`. These methods will also terminate if
    the callback raises ``StopIteration``.

    All methods except trust-constr (also) support a signature like::

        callback(xk)

    where ``xk`` is the current parameter vector.

    Introspection is used to determine which of the signatures above to
    invoke.

Returns
-------
res : OptimizeResult
    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.
```

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 :ref:`CG <optimize.minimize-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 :ref:`BFGS <optimize.minimize-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 :ref:`Newton-CG <optimize.minimize-newtoncg>` 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 :ref:`dogleg <optimize.minimize-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 :ref:`trust-ncg <optimize.minimize-trustncg>` 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 :ref:`trust-krylov <optimize.minimize-trustkrylov>` 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 :ref:`trust-exact <optimize.minimize-trustexact>`
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 iterations
and the most recommended for small and medium-size problems.

Bound-Constrained minimization

Method :ref:`Nelder-Mead <optimize.minimize-neldermead>` 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 :ref:`L-BFGS-B <optimize.minimize-lbfgsb>` uses the L-BFGS-B algorithm [6]_, [7]_ for bound constrained minimization.

Method :ref:`Powell <optimize.minimize-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 :ref:`TNC <optimize.minimize-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 :ref:`COBYLA <optimize.minimize-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 :ref:`COBYQA <optimize.minimize-cobyqa>` uses the Constrained Optimization BY Quadratic Approximations (COBYQA) method [18]_. The algorithm is a derivative-free trust-region SQP method based on quadratic approximations to the objective function and each nonlinear constraint. The bounds are treated as unrelaxable constraints, in the sense that the algorithm always respects them throughout the optimization process.

Method :ref:`SLSQP <optimize.minimize-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 :ref:`trust-constr <optimize.minimize-trustconstr>` is a trust-region algorithm for constrained optimization. It switches 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 switches 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 :ref:`trust-constr <optimize.minimize-trustconstr>`
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 handle 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. If the
gradient is estimated via finite-differences the Hessian must be
estimated using one of the quasi-Newton strategies.

Method specific options for the `hess` **keyword**

method/Hess	None	callable	'2-point/'3-point'/'cs'	HUS
Newton-CG	x	(n, n) LO	x	x
dogleg		(n, n)		
trust-ncg		(n, n)	x	x
trust-krylov		(n, n)	x	x
trust-exact		(n, n)		
trust-constr	x	(n, n) LO sp	x	x

where LO=LinearOperator, sp=Sparse matrix, HUS=HessianUpdateStrategy

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.

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] 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 Plantenga. 1998. On the
    implementation of an algorithm for large-scale equality constrained
    optimization. SIAM Journal on Optimization 8.3: 682-706.
.. [18] Ragonneau, T. M. *Model-Based Derivative-Free Optimization Methods
    and Software*. PhD thesis, Department of Applied Mathematics, The Hong
    Kong Polytechnic University, Hong Kong, China, 2022. URL:
    https://theses.lib.polyu.edu.hk/handle/200/12294.

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).

Help on function det in module scipy.linalg._basic:

det(a, overwrite_a=False, check_finite=True)
    Compute the determinant of a matrix

    The determinant is a scalar that is a function of the associated square
    matrix coefficients. The determinant value is zero for singular matrices.
```

```
Parameters
----------
a : (..., M, M) array_like
    Input array to compute determinants for.
overwrite_a : bool, optional
    Allow overwriting data in a (may enhance performance).
check_finite : bool, optional
    Whether to check that the input matrix contains only finite numbers.
    Disabling may give a performance gain, but may result in problems
    (crashes, non-termination) if the inputs do contain infinities or NaNs.

Returns
-------
det : (...) float or complex
    Determinant of `a`. For stacked arrays, a scalar is returned for each
    (m, m) slice in the last two dimensions of the input. For example, an
    input of shape (p, q, m, m) will produce a result of shape (p, q). If
    all dimensions are 1 a scalar is returned regardless of ndim.

Notes
-----
The determinant is computed by performing an LU factorization of the
input with LAPACK routine 'getrf', and then calculating the product of
diagonal entries of the U factor.

Even if the input array is single precision (float32 or complex64), the
result will be returned in double precision (float64 or complex128) to
prevent overflows.

Examples
--------
>>> import numpy as np
>>> from scipy import linalg
>>> a = np.array([[1,2,3], [4,5,6], [7,8,9]])  # A singular matrix
>>> linalg.det(a)
0.0
>>> b = np.array([[0,2,3], [4,5,6], [7,8,9]])
>>> linalg.det(b)
3.0
>>> # An array with the shape (3, 2, 2, 2)
>>> c = np.array([[[[1., 2.], [3., 4.]],
...                [[5., 6.], [7., 8.]]],
...               [[[9., 10.], [11., 12.]],
...                [[13., 14.], [15., 16.]]],
...               [[[17., 18.], [19., 20.]],
...                [[21., 22.], [23., 24.]]]])
>>> linalg.det(c)  # The resulting shape is (3, 2)
array([[-2., -2.],
       [-2., -2.],
       [-2., -2.]])
>>> linalg.det(c[0, 0])  # Confirm the (0, 0) slice, [[1, 2], [3, 4]]
-2.0
```

In [13]:
```
import scipy
help(scipy.linalg.det)
```

```
Help on function det in module scipy.linalg._basic:

det(a, overwrite_a=False, check_finite=True)
    Compute the determinant of a matrix

    The determinant is a scalar that is a function of the associated square
    matrix coefficients. The determinant value is zero for singular matrices.

    Parameters
    ----------
    a : (..., M, M) array_like
        Input array to compute determinants for.
    overwrite_a : bool, optional
        Allow overwriting data in a (may enhance performance).
    check_finite : bool, optional
        Whether to check that the input matrix contains only finite numbers.
        Disabling may give a performance gain, but may result in problems
        (crashes, non-termination) if the inputs do contain infinities or NaNs.

    Returns
    -------
    det : (...) float or complex
        Determinant of `a`. For stacked arrays, a scalar is returned for each
        (m, m) slice in the last two dimensions of the input. For example, an
        input of shape (p, q, m, m) will produce a result of shape (p, q). If
        all dimensions are 1 a scalar is returned regardless of ndim.

    Notes
    -----
    The determinant is computed by performing an LU factorization of the
    input with LAPACK routine 'getrf', and then calculating the product of
    diagonal entries of the U factor.

    Even if the input array is single precision (float32 or complex64), the
    result will be returned in double precision (float64 or complex128) to
    prevent overflows.

    Examples
    --------
    >>> import numpy as np
    >>> from scipy import linalg
    >>> a = np.array([[1,2,3], [4,5,6], [7,8,9]])  # A singular matrix
    >>> linalg.det(a)
    0.0
    >>> b = np.array([[0,2,3], [4,5,6], [7,8,9]])
    >>> linalg.det(b)
    3.0
    >>> # An array with the shape (3, 2, 2, 2)
    >>> c = np.array([[[[1., 2.], [3., 4.]],
    ...                [[5., 6.], [7., 8.]]],
    ...               [[[9., 10.], [11., 12.]],
    ...                [[13., 14.], [15., 16.]]],
    ...               [[[17., 18.], [19., 20.]],
    ...                [[21., 22.], [23., 24.]]]])
    >>> linalg.det(c)  # The resulting shape is (3, 2)
    array([[-2., -2.],
           [-2., -2.],
           [-2., -2.]])
    >>> linalg.det(c[0, 0])  # Confirm the (0, 0) slice, [[1, 2], [3, 4]]
    -2.0
```

Álgebra Lineal con SciPy (scipy.linalg)

Introdución a scipy.linalg

- El modulo scipy.linalg proporciona funciones avanzadas para operaciones con matrices, descomposiciones, sistema de ecuaciones lineales y mas.
- Extendiendo las capacidades básicas de álgebra lineal de NumPy (numpy.linalg)

Ventajas de scipy.linalg sobre numpy.linalg:

- ofrece más métodos avanzados y optimizados.
- Puede manejar matrices dispersas (grandes y con muchos ceros)
- Esta diseñado para cálculos científicos más complejos.

Operaciones Básicas con Matrices.

In [19]:
```
import numpy as np

matriz_a = np.array([[4,5],[3,2]])

vector_b = np.array([1,2])

print('Matriz:\n', matriz_a)
print('Vector:\n', vector_b)
```

```
Matriz:
 [[4 5]
 [3 2]]
Vector:
 [1 2]
```

Transpuesta de la Matriz

In [17]:
```
matriz_trans = matriz_a.T
print('Matriz transpuesta:\n', matriz_trans)
```

```
Matriz transpuesta:
 [[4 3]
 [5 2]]
```

Norma de una Matriz

In [12]:
```
from scipy.linalg import norm
```

In [18]:
```
from scipy.linalg import norm

norma_mat = norm(matriz_a)
print('Norma de la matriz:', norma_mat)
```

```
Norma de la matriz: 7.3484692283495345
```

In [9]:
```
a_norm = np.arange(9) - 4.0
a_norm
```

Out[9]:
```
array([-4., -3., -2., -1.,  0.,  1.,  2.,  3.,  4.])
```

In [10]:
```
b_norm = a_norm.reshape((3,3))
b_norm
```

Out[10]:
```
array([[-4., -3., -2.],
       [-1.,  0.,  1.],
       [ 2.,  3.,  4.]])
```

In [ ]:
```
norm(a_norm)
```

Out[ ]:
```
7.745966692414834
```

In [16]:
```
norm(b_norm)
```

Out[16]:
```
7.745966692414834
```

In [17]:
```
norm(a_norm, np.inf)
```

Out[17]:
```
4.0
```

Cálculo de Determinantes e Inversas de Matrices:

Cálculo del Determinante

In [20]:
```
from scipy.linalg import det

determinante_a = det(matriz_a)
print('Determinante de la matriz:', determinante_a)
```

```
Determinante de la matriz: -7.0
```

In [21]:
```
mat_a = np.array([[1,3,4], [4,5,6], [7,8,9]])
linalg.det(mat_a)
```

Out[21]:
```
2.9999999999999982
```

In [22]:
```
mat_c = np.array([[[[1., 2.], [3., 4.]],
...                [[5., 6.], [7., 8.]]],
...               [[[9., 10.], [11., 12.]],
...                [[13., 14.], [15., 16.]]],
...               [[[17., 18.], [19., 20.]],
...                [[21., 22.], [23., 24.]]]])

mat_c
```

Out[22]:
```
array([[[[ 1.,  2.],
         [ 3.,  4.]],

        [[ 5.,  6.],
         [ 7.,  8.]]],


       [[[ 9., 10.],
         [11., 12.]],

        [[13., 14.],
         [15., 16.]]],


       [[[17., 18.],
         [19., 20.]],

        [[21., 22.],
         [23., 24.]]]])
```

In [23]:
```
linalg.det(mat_c)
```

Out[23]:
```
array([[-2., -2.],
       [-2., -2.],
       [-2., -2.]])
```

Inversa de una Matriz

In [24]:
```
from scipy.linalg import inv

inversa_mat = inv(matriz_a)
print('Inversa de la matriz:', inversa_mat)
```

```
Inversa de la matriz: [[-0.28571429  0.71428571]
 [ 0.42857143 -0.57142857]]
```

Verificación de la inversa (matriz_a * inversa_mat = identidad)

In [22]:
```
identidad = matriz_a @ inversa_mat
print('Matriz identidad resultante:', identidad)
```

```
Matriz identidad resultante: [[ 1.00000000e+00 -2.22044605e-16]
 [ 3.33066907e-16  1.00000000e+00]]
```

In [26]:
```
inv_a = np.array([[1.,2.],[3.,4.]])
inv_a
```

Out[26]:
```
array([[1., 2.],
       [3., 4.]])
```

In [27]:
```
linalg.inv(inv_a)
```

Out[27]:
```
array([[-2. ,  1. ],
       [ 1.5, -0.5]])
```

Resolución de Sistemas de Ecuaciones Lineales

- Resolver el sistema de ecuaciones:

$4x + 7y = 10$

$2x + 6y = 8$

- Representación Matricial del sistema

Ax = b

- Donde A es la matriz de coeficientes.

$$A = \begin{bmatrix} 4 & 7 \\ 2 & 6 \end{bmatrix}$$

- x es el vector incógnita

$$x = \begin{bmatrix} x \\ y \end{bmatrix}$$

- b es el vector de resultados

$$b = \begin{bmatrix} 10 \\ 8 \end{bmatrix}$$

- Usando scpi.linalg.solve()

In [3]:
```
from scipy.linalg import solve

A0 = np.array([[4, 7], [2, 6]])
b0 = np.array([10, 8])


x = solve(A0, b0)
print('Solución del Sistema')
print('x =',x[0])
print('y =',x[1])
```

```
Solución del Sistema
x = 0.3999999999999999
y = 1.2
```

Ejemplo con 3 incógnitas

- Resolver el sistema de ecuaciones:

$2x + y - z = 8$

$-3x + 2y + 4z = -11$

$-2x + y + 2z = -3$

- Representación Matricial del sistema

Ax = b

- Donde A es la matriz de coeficientes.

$$A = \begin{bmatrix} 2 & 1 & -1 \\ -3 & 2 & 4 \\ -2 & 1 & 2 \end{bmatrix}$$

- x es el vector incógnita

$$x = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

- b es el vector de resultados

$$b = \begin{bmatrix} 8 \\ -11 \\ -3 \end{bmatrix}$$

- Usando scpi.linalg.solve()

In [4]:
```
from scipy.linalg import solve

A1 = np.array([[2,1,-1],[-3,2,4],[-2,1,2]])
b1 = np.array([8,-11,-3])

x = solve(A1,b1)
```

```
print('Solución del sistema')
print('x =', x[0])
print('y =', x[1])
print('z =', x[2])
```

```
Solución del sistema
x = -5.000000000000001
y = 7.666666666666667
z = -10.333333333333336
```

Valores Propios y Vectores Propios

- Dado que un sistema de ecuaciones de la forma Ax = λx, los valores λ son los valores propios, y los vactores x correspondientes son los vectores propios:
- Ejemplo con una matriz de 2*2

In [28]:
```
import numpy as np
from scipy.linalg import eig
```

In [ ]:
```
matriz = np.array([[4, 2], [1, 3]])

valores_propios, vec_propios = eig(matriz)

print('Valores propios: \n', valores_propios)
print('Vectores propios: \n', vec_propios)
```

```
Valores propios:
 [5.+0.j 2.+0.j]
Vectores propios:
 [[ 0.89442719 -0.70710678]
 [ 0.4472136   0.70710678]]
```

In [29]:
```
eig_a = np.array([[0., -1.], [1., 0.]])
eig_a
```

Out[29]:
```
array([[ 0., -1.],
       [ 1.,  0.]])
```

In [30]:
```
eig(eig_a)
```

Out[30]:
```
(array([0.+1.j, 0.-1.j]),
 array([[0.70710678+0.j        , 0.70710678-0.j        ],
        [0.        -0.70710678j, 0.        +0.70710678j]]))
```

In [32]:
```
eig_b = np.array([[3.,0.,0.],[0.,8.,0.],[0.,0.,7.]])
eig_b
```

Out[32]:
```
array([[3., 0., 0.],
       [0., 8., 0.],
       [0., 0., 7.]])
```

In [33]:
```
eig(eig_b)
```

Out[33]:
```
(array([3.+0.j, 8.+0.j, 7.+0.j]),
 array([[1., 0., 0.],
        [0., 1., 0.],
        [0., 0., 1.]]))
```

Factorización y Descomposiciones de Matrices.

- LU (Factorización de matrices)

- Permite resolver ecuaciones de forma más eficiente al dividir la matriz en dos matrices triangulares. A = LU
- P -matriz de Permutación
- L Triangular inferior
- U Triangular Superior.

In [34]:
```
from scipy.linalg import lu
```

In [ ]:
```
A3 = np.array([[3,2],[1,4]])
P, L, U = lu(A3)

print('Matriz P (Permutacion)\n', P)
print('Matriz L (Triangular inferior)\n', L)
print('Matriz U (Triangular superior)\n', U)
```

```
Matriz P (Permutacion)
 [[1. 0.]
 [0. 1.]]
Matriz L (Triangular inferior)
 [[1.         0.        ]
 [0.33333333 1.        ]]
Matriz U (Triangular superior)
 [[3.         2.        ]
 [0.         3.33333333]]
```

Demostración con una Matriz 4D

In [35]:
```
rango_mat = np.random.default_rng()
A4 = rango_mat.uniform(low = -4, high=4, size=[3,2,4,8])
A4
```

Out[35]:
```
array([[[[-2.64503087, -1.03991414,  1.947357  ,  2.23591602,
          -1.37114908, -3.8299527 , -0.40682926, -3.1089618 ],
         [ 0.41702575, -1.76826123, -2.47887403,  1.19282849,
           3.86413252,  0.35978076,  2.49906483,  3.76794597],
         [ 3.67663139,  0.81973242, -0.40764984, -0.37088772,
          -2.56917259,  2.49447467,  3.99041204, -1.78730029],
         [-1.4489629 , -3.97202784, -3.7831243 , -3.09584089,
          -3.53012266, -2.82094656, -0.78533047, -2.19322522]],

        [[ 3.99404751,  0.02507099, -0.99397051, -1.91737177,
           0.58576091,  0.15002405,  3.19446122,  0.22969571],
         [ 3.75538978, -1.92775957,  1.29008218, -0.28886784,
           2.25426911, -1.01672023, -2.37415317,  1.64365153],
         [-0.67684241,  2.4271523 ,  2.04861311, -1.22409553,
          -3.82392756,  3.63048298, -0.75719709,  2.11941658],
         [-2.48705896, -0.12362866,  3.35095653, -3.68485387,
          -3.68342339,  1.3238732 ,  0.04063066,  2.47733251]]],


       [[[ 2.47461583,  0.25977928, -3.87484702, -3.26152579,
          -2.15957034,  3.61530699,  3.05414997,  3.26302049],
         [ 3.23333813, -3.82077311, -0.51553085, -1.02461157,
           2.50615595, -3.61444204, -1.87348796, -2.03066474],
         [ 1.71526208, -2.14483403, -1.8975361 , -1.22565841,
          -0.82948741,  1.90471275, -2.36445366, -2.404724  ],
         [-2.57726708,  1.25828025,  3.56542904,  3.21558479,
           0.28980806,  2.36215118,  2.54133656, -2.01010915]],

        [[-0.92093147,  1.50474865, -2.93692856,  1.528817  ,
          -3.11996138, -3.06644671, -3.4566753 ,  2.25849706],
         [ 0.78192341,  3.67800403,  3.88196787, -3.47898598,
          -2.54767529,  0.29050399, -2.78862135,  0.96185575],
         [ 2.68743032, -3.56370445,  3.32471407, -2.26081418,
          -3.77607252, -1.25831902,  1.88733566,  1.87256874],
         [-3.580211  ,  3.65432457, -1.5828499 ,  0.3662699 ,
           2.0989842 , -3.52837864, -3.56067068, -0.50194572]]],


       [[[-0.50371224, -2.91001218,  3.30913431,  0.99390534,
          -2.86476374, -3.6375522 , -1.07426264, -0.84166706],
         [ 0.61217241, -0.10580805, -2.35314605,  2.31090698,
          -0.96888484, -1.62147845,  1.71319414, -2.81791358],
         [ 2.22601605,  1.48177292, -0.40749145,  3.1608009 ,
           2.85202217,  2.77550818, -3.84387182,  0.5480975 ],
         [ 2.82187139,  3.4557871 , -3.79781154, -0.96235866,
           0.92363526, -3.25869291, -2.09041963, -2.86836204]],

        [[ 2.0271894 ,  2.63334414,  1.17779987,  0.92524022,
          -2.74532954, -3.793212  ,  3.20606219, -1.96540214],
         [-3.63614725, -3.13792609, -2.29244876,  2.67191936,
           0.76810189, -1.35175887, -2.60571338, -2.53490513],
         [ 1.99385054, -3.6118132 , -3.45650574,  0.82815199,
           2.06314523, -2.27773699,  2.41338421, -1.09544263],
         [ 3.04123191,  2.5247201 , -2.43172101,  0.97896314,
           1.85530266, -1.01404248, -1.51436268, -0.29031879]]]])
```

In [37]:
```
p, l , u = lu(A4)
p.shape, l.shape, u.shape
```

Out[37]:
```
((3, 2, 4, 4), (3, 2, 4, 4), (3, 2, 4, 8))
```

Factorización QR (Descomposición ortogonal)

- Se usa en regresión lineal y análisis de datos.
- Q es unitario
- R Triangular superior

In [38]:
```
from scipy.linalg import qr
```

In [ ]:
```
A4 = np.array([[12,-51,4],[6,167,-68],[-4,24,-41]])

Q, R = qr(A4)

print('Matriz Q: \n', Q)
print('Matriz R: \n', R)
```

```
Matriz Q:
 [[-0.85714286  0.39428571  0.33142857]
 [-0.42857143 -0.90285714 -0.03428571]
 [ 0.28571429 -0.17142857  0.94285714]]
Matriz R:
 [[ -14.  -21.   14.]
 [   0. -175.   70.]
 [   0.    0.  -35.]]
```

In [39]:
```
rng = np.random.default_rng()
rng_a = rng.standard_normal((9,6))
rng_a
```

Out[39]:
```
array([[-0.26657195,  0.35680844, -1.22233686, -0.65660046, -2.36364495,
        -0.25021945],
       [-0.2484332 , -1.10639876, -0.83053394,  1.12952238, -0.21719895,
        -0.8070626 ],
       [ 0.12447301, -1.36352883, -0.51189537,  1.57252638, -1.45802442,
         0.19184743],
       [-0.54053337,  1.27867035,  1.22541238,  0.70338968,  0.94413341,
        -0.85077738],
       [-0.69221404,  0.73768583,  0.61822354, -0.3197522 , -0.23673528,
        -0.68201496],
       [ 0.42361818, -0.84006075, -0.04111216,  2.39065032,  0.08614175,
        -0.94759063],
       [ 0.05824551, -0.71250356,  1.11493959, -0.92093627,  0.92114081,
         1.39470211],
       [ 0.62851903, -1.09131341, -0.82431312,  0.3024738 , -0.71341313,
        -0.05871281],
       [-0.23661428,  0.03312263, -0.14066094, -0.65499244, -0.51980229,
         0.2522772 ]])
```

In [43]:
```
q, r = linalg.qr(rng_a)
np.allclose(rng_a, np.dot(q,r))
q.reshape, r.reshape
```

Out[43]:
```
(<function ndarray.reshape>, <function ndarray.reshape>)
```

Factorización SVD (Singular Value Descomposition)

- Muy utilizada en Machine Learning, reducción de dimensiones y compresión de datos.

In [12]:
```
from scipy.linalg import svd

A5 = np.array([[1,2],[3,4],[5,6]])
U, S, Vh = svd(A5)

print('Matriz U: \n', U)
```

```
print('Matriz S: \n', S)
print('Matriz Vh: \n', Vh)
```

```
Matriz U:
 [[-0.2298477   0.88346102  0.40824829]
 [-0.52474482  0.24078249 -0.81649658]
 [-0.81964194 -0.40189603  0.40824829]]
Matriz S:
 [9.52551809 0.51430058]
Matriz Vh:
 [[-0.61962948 -0.78489445]
 [-0.78489445  0.61962948]]
```

Optimización con SciPy (scipy.optimize)

- La optimización es el proceso de encontrar el mínimo o máximo de una función.
- SciPy nos ofrece varias herramientas para optimización de funciones y resolución de ecuaciones.

Aplicaciones comunes de optimización:

- Ajuste de modelos en Machine Learning.
- Minimización de costos en ingeniería y negocios.
- Optimización de portafolios financieros.
- Ajuste de curvas en ciencia de datos.

Funciones principales de scipy.optimize

- minimiZe(): Minimización de funciones.
- root(): Solución de ecuaciones.
- linprog(): Programación lineal.
- curvefit(): Ajuste de curvas.

Minimización de funciones.

- Supongamos que queremos encontrar el mínimo de la función.

$f(x) = x^2 + 4x + 4$

- Paso 1: Definir la función en Python
- Paso 2: Usar scipy.optimize.minimize() para encontrar el mínimo.

In [2]:
```
import numpy as np
import scipy
from scipy.optimize import minimize
```

In [3]:
```
def funcion(x):
    return x**2 + 4*x +4

x0 = np.array([0])

resultado = minimize(funcion, x0)

print('Valor de x: ', resultado.x[0])
print('Valor minimo de la funcion  ', resultado.fun)
```

```
Valor de x:  -2.00000001888464
Valor minimo de la funcion   0.0
```

Minimización de la función:

$f(x) = x^4 - 3 * x^3 + 2$

In [7]:
```python
def funcion1(x):
    return x**4 - 3*x**3 +2

x1 = np.array([0])

resultado1 = minimize(funcion1, x1)

print('Valor de x:', resultado1.x[0])
print('Valor minimo de la funcion: ', resultado1.fun)
```

```
Valor de x: 0.0
Valor minimo de la funcion:  2.0
```

Solución de Ecuaciones con scipy.optimize.root()

- Vamos a resolver la ecuación
  - $x^2 - 4 = 0$
- Esta ecuación tiene dos soluciones.
  - x = 2 y x = -2

In [14]:
```python
from scipy.optimize import root

def ecuacion(x):
    return x**2 - 4

x0 = np.array([1]) # <--- Cambias el valor inical para obtener otra solucion

solucion = root(ecuacion , x0)

print('Solucion de la ecuacion: ', solucion.x[0])
```

```
Solucion de la ecuacion:  2.0000000000000004
```

Programación Lineal con scipy.optimize.linprog()

- Maximizar una función de beneficios. Ejemplo:
- Un empresario produce dos productos.
  - Producto A --> Beneficio de $3 por unidad.
  - Producto B --> Beneficio de $5 por unidad.
- Debe decidir cuántos productos productos producir, sujeto a estas restricciones.

1. Tiempo de producción: 1 hora para A y 2 horas para B (máximo de 6 horas disponibles)
2. Material disponible: 1 unidad para A y 1 unidad para B (máximo 4 unidades)

Definimos el problema en programación lineal:

$MaxzimizarZ = 3x + 5y$

Donde:

$x + 2y <= 6$

$x + y <= 4$

$x >= 0, y >= 0$

In [18]:
```
from scipy.optimize import linprog

c = [-3, -5]

A = [[1,2],[1,1]]
b = [6,4]

x_bounds = (0, None)
y_bounds = (0, None)

resultado = linprog(c, A_ub=A, b_ub=b, bounds=[x_bounds, y_bounds], method='highs')

print('Cantidad optima del Producto A:', resultado.x[0])
print('Cantidad optima del Producto B:', resultado.x[1])
print('Beneficio Maximo:', -resultado.fun)
```

```
Cantidad optima del Producto A: 2.0
Cantidad optima del Producto B: 2.0
Beneficio Maximo: 16.0
```

Ajuste de curvas con scipy.optimize.curve_fit()

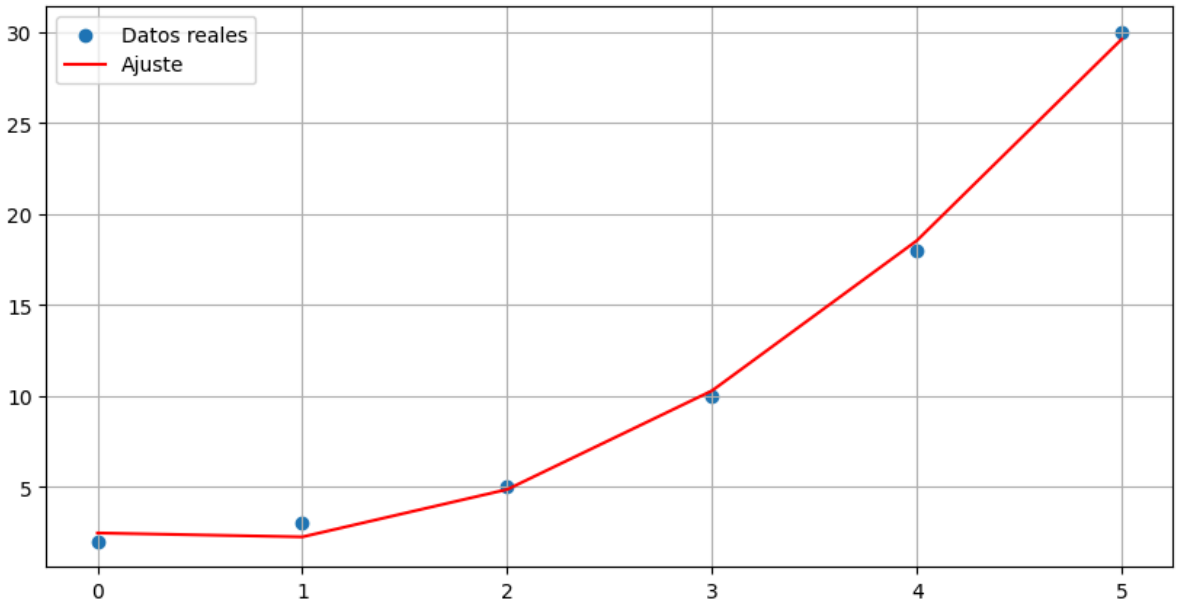
Ajustar una curva a datos experimentales.

- Supongamos que tenemos datos experimentales y queremos ajustar una función.

$y = ax^2 + bx + c$

In [20]:
```
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit
```

In [22]:
```
x_datos = np.array([0,1,2,3,4,5])
y_datos = np.array([2,3,5,10,18,30])


def modelo(x, a, b, c):
    return a*x**2 + b*x + c

parametros, _ = curve_fit(modelo, x_datos, y_datos)

plt.figure(figsize=(10,5))
plt.scatter(x_datos, y_datos, label='Datos reales')
plt.plot(x_datos, modelo(x_datos, *parametros), label='Ajuste', color='red')
plt.legend()
plt.grid(True)
plt.show()
```



Estadística y Probabilidad con SciPy (scipy.stats)

- SciPy proporciona herramientas avanzadas para el análisis estadístico, incluyendo funciones para:
  - Distribuciones de probabilidad (teóricas y empíricas)
  - Pruebas estadísticas (t-test, chi-cuadrado, etc)
  - Generación de números aleatorios.

Medidas Estadísticas Básicas

- Podemos calcular estadísticas descriptivas como la media, media, moda, varianza, desviación estándar, entre otras.

In [28]:
```
import numpy as np
from scipy import stats

datos = np.array([1,3,7,9,8,4,6,2,5,9,7,1,3,6,8,9,2,5,7,4])

media = np.mean(datos)
mediana = np.median(datos)
moda = stats.mode(datos)
varianza = np.var(datos, ddof=1)
desviacion = np.std(datos, ddof=1)

print('Calculo de Estadísticas Básicas: \n')
print(f'Media: {media}')
print(f'Mediana: {mediana}')
print(f'Moda: {moda.mode} (frecuencia: {moda.count})')
print(f'Varianza: {varianza}')
print(f'Desviación Estándar: {desviacion}')
```

```
Calculo de Estadísticas Básicas:

Media: 5.3
Mediana: 5.5
Moda: 7 (frecuencia: 3)
Varianza: 7.273684210526315
Desviación Estándar: 2.696976865033572
```

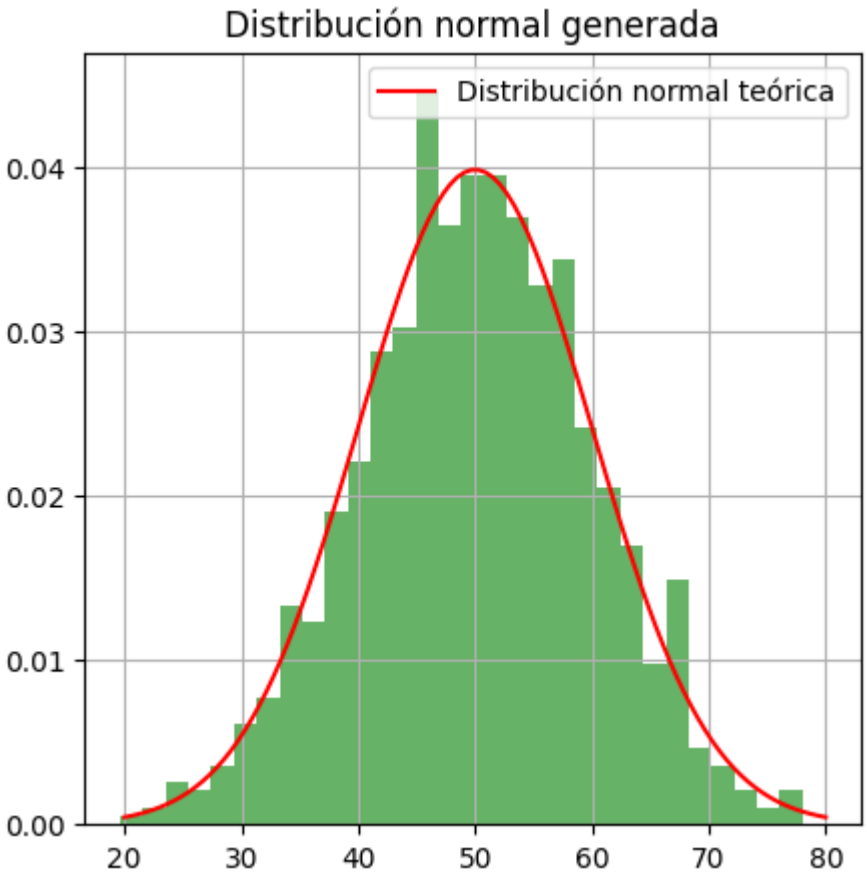
Generación de Números Aleatorios.

- Podemos generar datos aleatorios siguiendo una distribución especifica con scipy.stats

rvs(loc=μ, scale=σ, size=N): Genera N valores con media μ y desviación σ

pdf(x) : Calcula la función densidad de probabilidad (PDF) para los valores x

In [31]:
```
import matplotlib.pyplot as plt

data = stats.norm.rvs(loc=50, scale=10, size=1000)

plt.figure(figsize=(5,5))
plt.hist(data, bins=30, density=True, alpha=0.6, color='g')

x = np.linspace(20, 80, 100)
pdf = stats.norm.pdf(x, loc=50, scale=10)
plt.plot(x, pdf, 'r-', label='Distribución normal teórica')
plt.legend()
plt.title('Distribución normal generada')
plt.grid(True)
plt.show()
```



Pruebas de Normalidad

- Las pruebas de normalidad determinan si un conjunto de datos sigue una distribución normal

In [32]:
```
stat, p = stats.shapiro(data)
print(f'Estadistico de prueba: {stat}, p-valor: {p}')
```

```
if p > 0.05:
    print('Los datos parecen seguir una distribución normal.')
else:
    print('Los datos no siguen una distribución normal')
```

```
Estadistico de prueba: 0.9988701638407441, p-valor: 0.7993161760313972
Los datos parecen seguir una distribución normal.
```

Pruebas Estadísticas:

Prueba	Usada para...
ttest_1samp()	Comparar la media de una muestra con un valor especifico.
ttest_ind()	Comparar la media de dos muestras independientes
ttest_rel()	Compara la media de dos muestras dependientes (pareadas)
chi2_contingency()	Prueba de independencia Chi-cuadrado

Prueba t para una muestra.

- Se usa cuando queremos comprar la media de una muestra con un valor especifico.

In [35]:
```
t_stat, p_valor = stats.ttest_1samp(data, 50)
print(f'Estadistico t: {t_stat}, p_valor: {p_valor}')

if p_valor > 0.05:
    print('La media es diferente de 50')
else:
    print('La media es diferente de 50')
```

```
Estadistico t: 0.19077414746118618, p_valor: 0.8487413054960656
La media es diferente de 50
```

Correlación entre Variables

- La correlación mide la relación entre dos variables.
- Coeficientes más comunes:
  - Coeficiente de Pearson (lineal): stats.pearsonr(x, y)
  - Coeficiente de Spearman (monótona): stats.spearmenr(x, y)

In [36]:
```
x = np.array([1,2,3,4,5,6,7,8,9])
y  = np.array([10,12,13,14,15,16,17,18,19])

corr, p_value = stats.pearsonr(x, y)

print(f'Coeficiente de correlación Pearson : {corr} ')
print(f'p-valor: {p_value}')

if p_value < 0.05:
    print("Existe una correlación estadísticamente significativa.")
else:
    print("No hay evidencia de correlación significativa.")
```

```
Coeficiente de correlación Pearson : 0.9954736269412081
p-valor: 2.045269433273381e-08
Existe una correlación estadísticamente significativa.
```

Procesamiento de Señales con SciPy (scipy.singnal)

- El modulo scipy.signal proporciona herramientas para filtrado, análisis espectral, convolución, detección de pico, generación de señales y mas.

Aplicaciones:

- Filtrado de ruido en señales de audio.
- Procesamiento de imágenes y video.
- Detección de patrones en datos.
- Análisis de señales en medicina

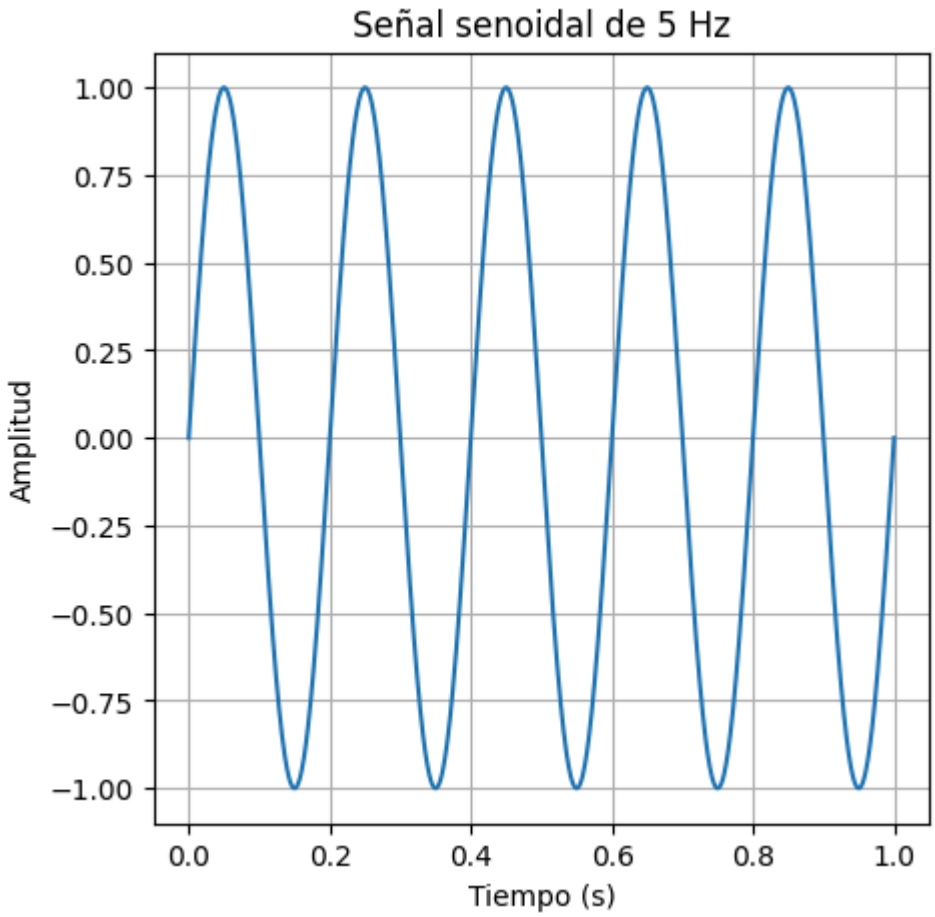
Generación de Señales.

- Podemos generar señales de diferentes formas, como senoidales, cuadradas y triangulares

In [37]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import chirp
```

Creamos una señal senoidal de 5 Hz durante 1 seg, con 1000 muestras por segundo. np.sin(2π*frecuencia_inicial*tiempo) Genera la señal senoidal con frecuencia_muestro

In [47]:
```
frecuencia_muestreo = 1000
tiempo = np.linspace(0, 1, frecuencia_muestreo)
frecuencia_inical = 5

señal = np.sin(2 * np.pi * frecuencia_inical * tiempo)

plt.figure(figsize=(5,5))
plt.plot(tiempo, señal)
plt.title('Señal senoidal de 5 Hz')
plt.xlabel('Tiempo (s)')
plt.ylabel('Amplitud')
plt.grid()
plt.show()
```



Transformada de Fourier con SciPy

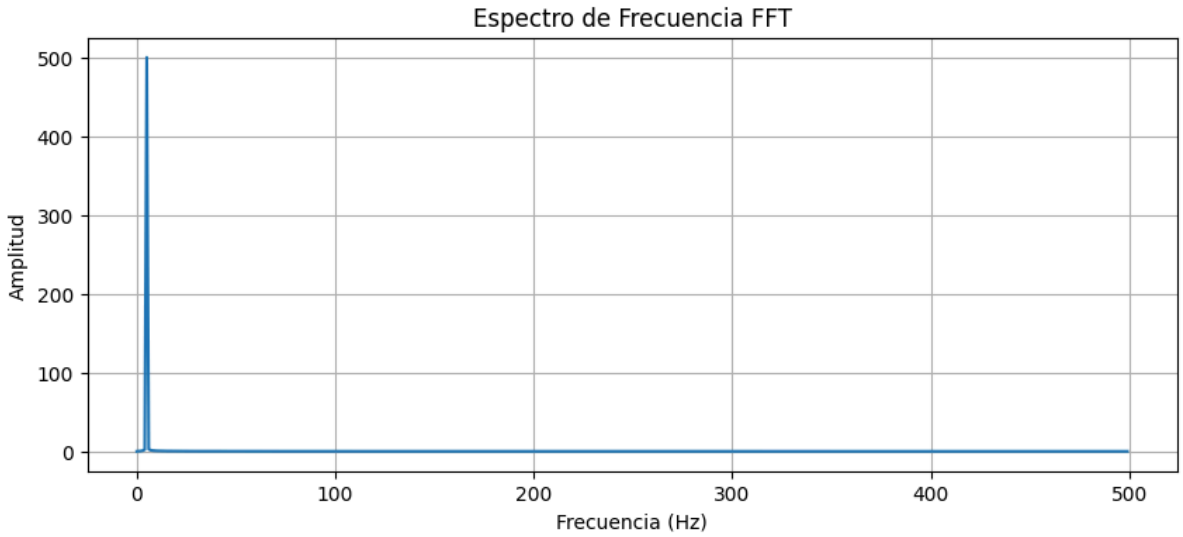
- La transformada Rápida de Fourier (FFT) nos permite analizar la frecuencia de una señal

fft(señal): calcula la FFT de la señal.

fftfreq(N, 1/frecuencia_muestreo): Devuelve las frecuencia s correspondientes.

Gráficamos solo la mitad positiva (N // 2).

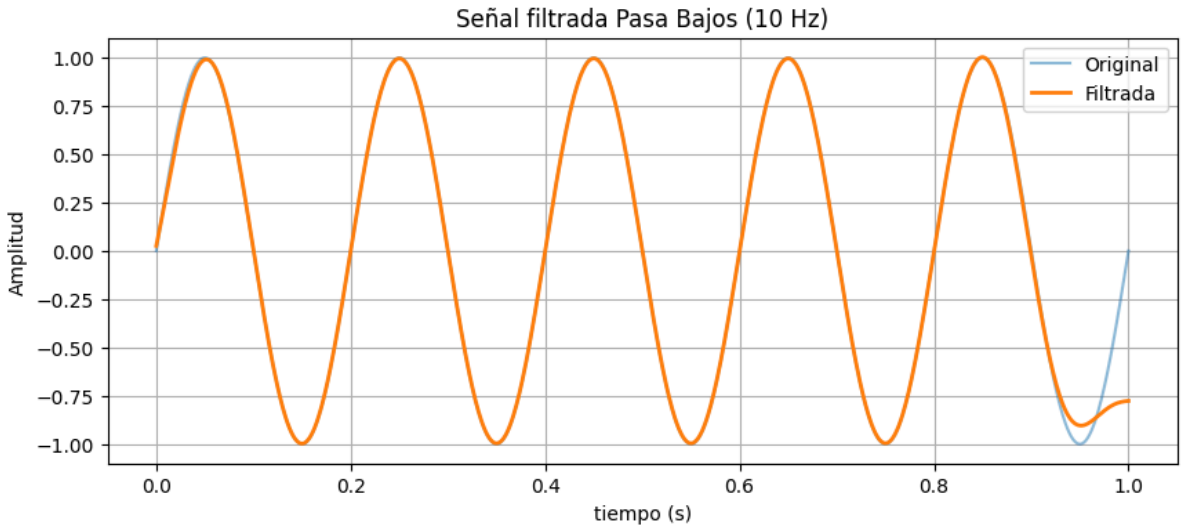
In [48]:
```
from scipy.fft import fft, fftfreq

N = len(señal)
frecuencias = fftfreq(N, 1/frecuencia_muestreo)
espectro = np.abs(fft(señal))

plt.figure(figsize=(10,4))
plt.plot(frecuencias[:N//2], espectro[:N//2])
plt.title('Espectro de Frecuencia FFT')
plt.xlabel('Frecuencia (Hz)')
plt.ylabel('Amplitud')
plt.grid(True)
plt.show()
```



Filtros Digitales.

- Podemos aplicar filtros pasa bajas, pasa altas y pasa bandas a las señales.

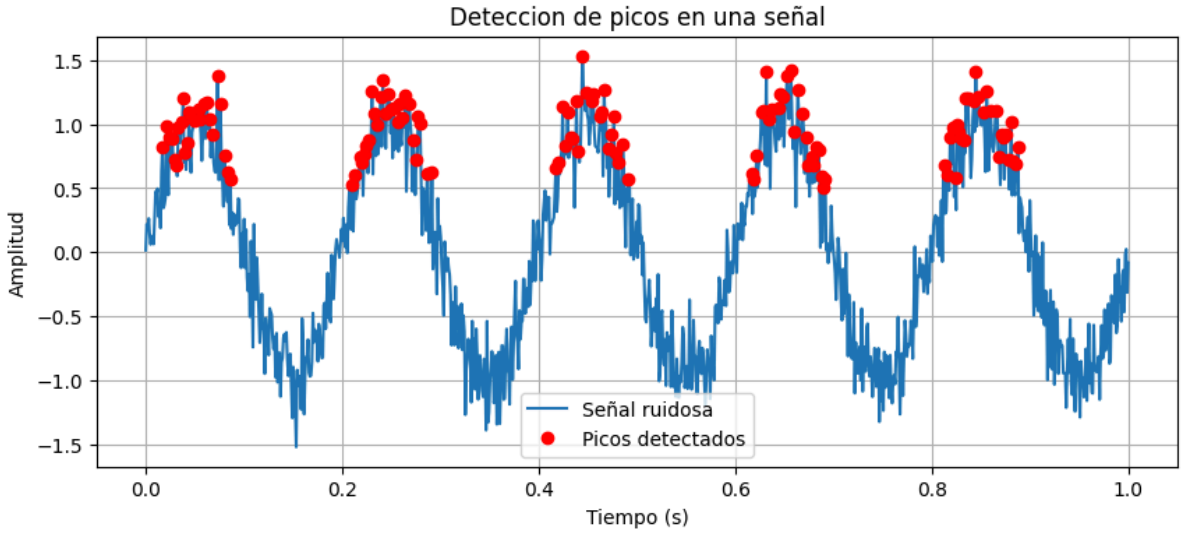
In [53]:
```
from scipy.signal import butter, filtfilt

fs=1000
frecuencia_corte = 10
b, a = butter(N=4, Wn=frecuencia_corte, fs=fs, btype='low')

señal_filtrada = filtfilt(b, a, señal)

plt.figure(figsize=(10,4))
plt.plot(tiempo, señal, label='Original', alpha=0.5)
plt.plot(tiempo, señal_filtrada, label='Filtrada', linewidth=2)
plt.title('Señal filtrada Pasa Bajos (10 Hz)')
plt.xlabel('tiempo (s)')
plt.ylabel('Amplitud')
plt.legend()
plt.grid()
plt.show()
```



Detección de Picos

- Detectamos picos en una señal, útil en ECG, audio y detección de eventos

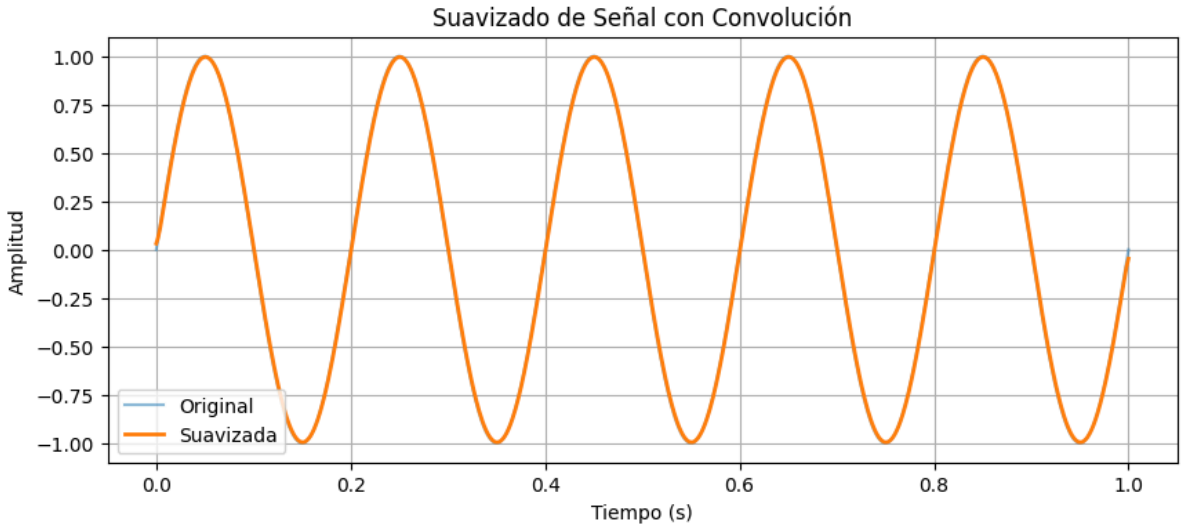
In [56]:
```
from scipy.signal import find_peaks


señal_ruidosa = señal + np.random.normal(0, 0.2, len(señal))

picos, _  = find_peaks(señal_ruidosa, height=0.5)



plt.figure(figsize=(10,4))
plt.plot(tiempo, señal_ruidosa, label='Señal ruidosa')
plt.plot(tiempo[picos], señal_ruidosa[picos], 'ro', label='Picos detectados')
plt.title('Deteccion de picos en una señal')
plt.xlabel('Tiempo (s)')
plt.ylabel('Amplitud')
plt.legend()
plt.grid(True)
plt.show()
```



Convolución de Señales.

- La convolución se usa para visualizar o aplicar filtros.

In [58]:
```
from scipy.signal import convolve


ventana = np.ones(10) / 10


señal_suavizada = convolve(señal, ventana, mode='same')

plt.figure(figsize=(10,4))
plt.plot(tiempo, señal, label="Original", alpha=0.5)
plt.plot(tiempo, señal_suavizada, label="Suavizada", linewidth=2)
plt.title("Suavizado de Señal con Convolución")
plt.xlabel("Tiempo (s)")
plt.ylabel("Amplitud")
plt.legend()
plt.grid()
plt.show()
```



Integración y Ecuaciones Diferenciales (scipy.integrate, scipy.odeint)

- El modulo scipy.integrate proporciona funciones para:
  - Integración de funciones.
  - Integrales definidas e indefinidas.
  - Ecuaciones diferenciales ordinarias (ODEs)

Integracion de funciones.

- SciPy permite calcular la integral definida de una función f(x) en un intervalo [a, b].

Integral definida de una función usando quad:

- Calcular la integral de f(x) en el intervalo [0,2]
- quad(f, a,b) Calcula la integral de f(x) en [a,b]

In [44]:
```
import numpy as np
from scipy.integrate import quad
```

In [ ]:
```
def f(x):
    return x**2

resultado, error = quad(f, 0, 2)

print(f'Integral de x^2 en [0,2]: {resultado:.4f}')
print(f'Error estimado: {error: .4e}')
```

```
Integral de x^2 en [0,2]: 2.6667
Error estimado:  2.9606e-14
```

Integral Doble con dblquad

- Para integrales dobles usamos dblquad
- dblquad(f, ax, bx, gfun, hfun)
- Integra f(x, y) en [ax, bx] y [gfun(x), hfun(x)]

Calcular la integral en region : 0 <= y <= 2, 0 <= x <=3

In [66]:
```
from scipy.integrate import dblquad

def f1(x, y):
    return x * y

resultado1, error1 = dblquad(f1, 0, 3, lambda x: 0, lambda x: 2)

print(f'Integral doble de x*y en [0,3]x[0,2]: {resultado1:.4f}')
print(f'Error estimado: {error1:.4e}')
```

```
Integral doble de x*y en [0,3]x[0,2]: 9.0000
Error estimado: 9.9920e-14
```

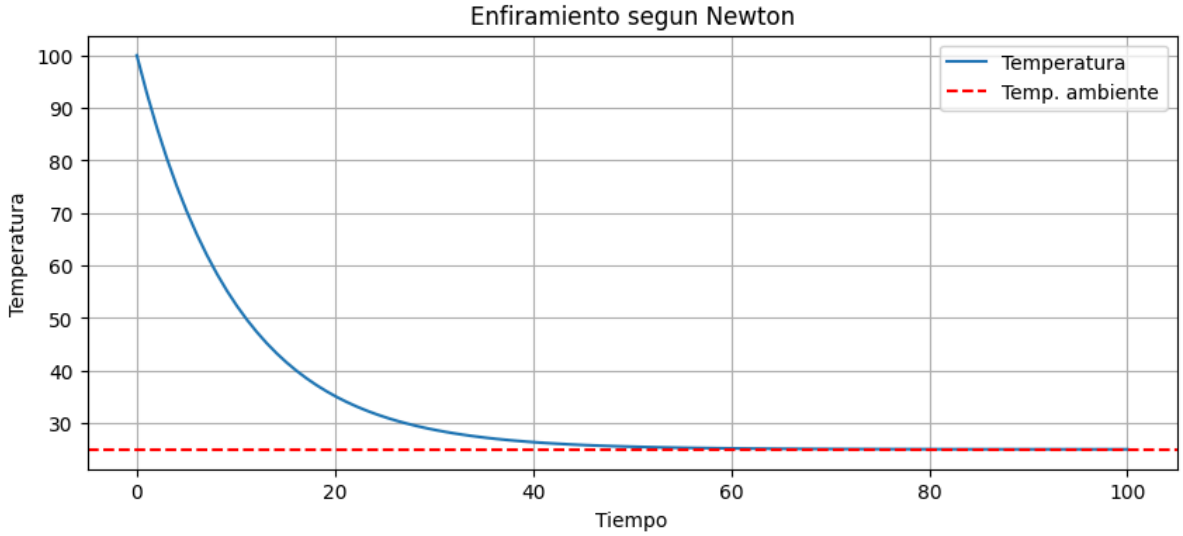
Resolución de Ecuaciones Diferenciales Ordinales (ODEs)

- Las ecuaciones diferenciales modelan sistemas físicos y biológicos.
- SciPy proporciona odeint para resolverlas.

Ecuación diferencial de una sistema de enfriamiento.

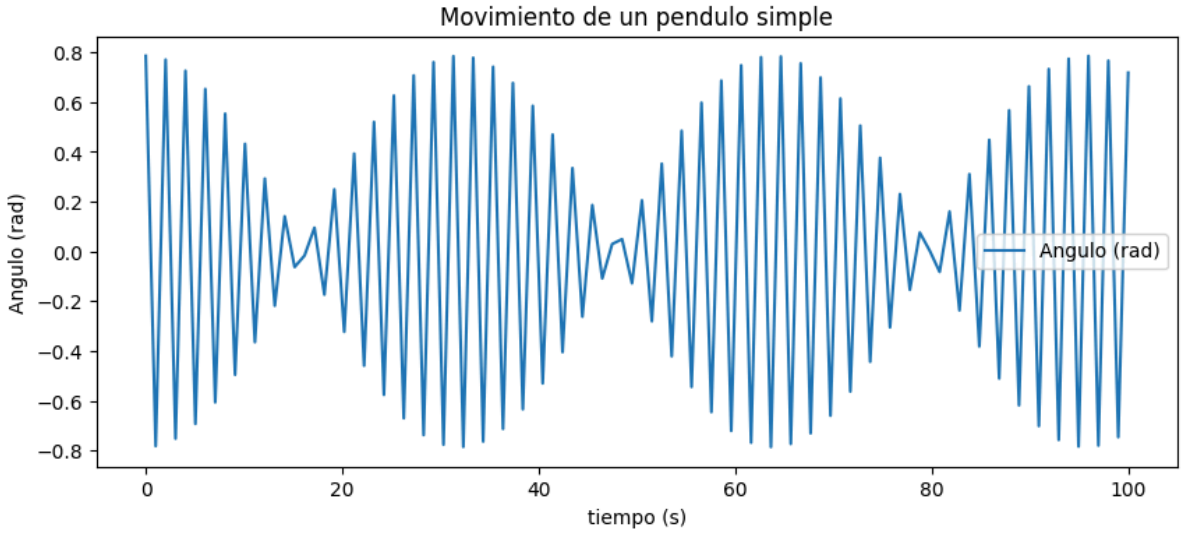
- (Modelo de Newton: dy/dt = -k(y - T_amb))
- modelo(y, t. K, T_amb) Ecuacion Diferencial
- odeint(modelo, y0, t, args(k, T_amb)) Resolver ODE
  - resuelve dy/dt = -k(y -T_amb) con y0 = 100 y k= 0.1

In [68]:
```
from scipy.integrate import odeint
import matplotlib.pyplot as plt

def modelo(y, t, k, T_amb):
    return -k * (y -T_amb)

y0 = 100
k = 0.1
T_amb =25
t = np.linspace(0, 100, 100)

solucion = odeint(modelo, y0, t, args=(k, T_amb))

plt.figure(figsize=(10,4))
plt.plot(t, solucion, label=('Temperatura'))
plt.axhline(T_amb, linestyle='--', color='red', label='Temp. ambiente')
plt.title('Enfiramiento segun Newton')
plt.xlabel('Tiempo')
plt.ylabel('Temperatura')
plt.legend()
plt.grid(True)
plt.show()
```



Ecuación Diferencial de un péndulo simple.

In [69]:
```
def pendulo(theta, t1, L, g):
    theta1, theta2 = theta
    dtheta1_dt = theta2
    dtheta2_dt = -(g /L)* np.sin(theta1)
    return [dtheta1_dt, dtheta2_dt]

L = 1
g = 9.81
theta0 = [np.pi / 4, 0]
t1 = np.linspace(0, 10, 250)

solucion1 = odeint(pendulo, theta0, t, args=(L, g))

plt.figure(figsize=(10,4))
plt.plot(t, solucion1[:, 0], label='Angulo (rad)')
plt.title('Movimiento de un pendulo simple')
plt.xlabel('tiempo (s)')
plt.ylabel('Angulo (rad)')
plt.legend()
plt.show()
```

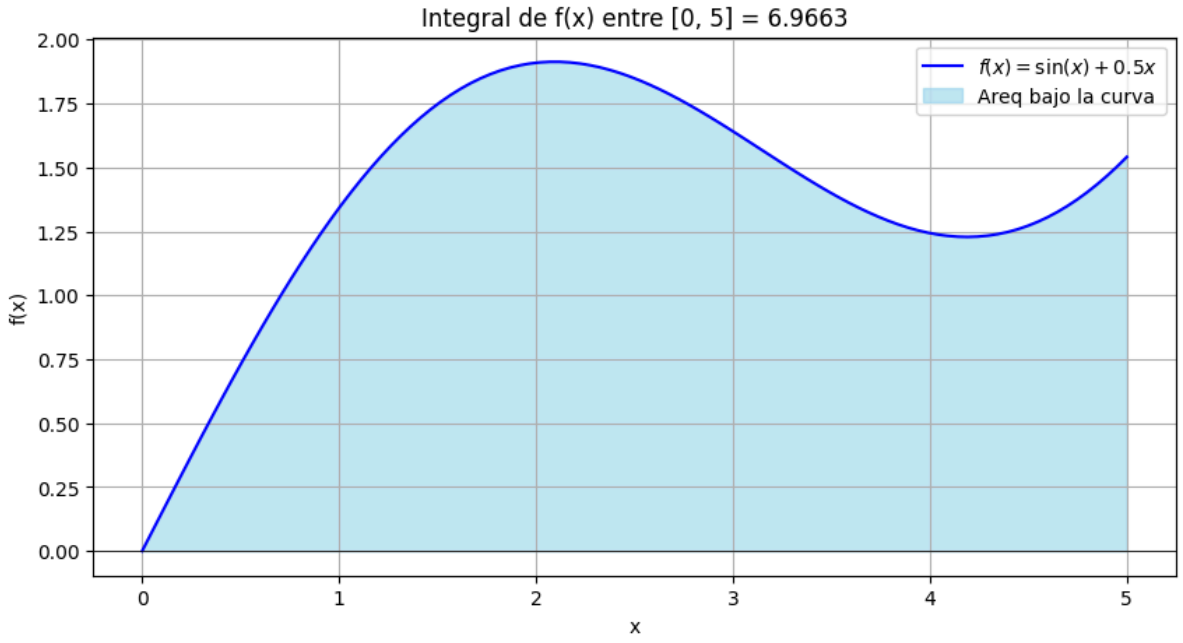


Integral de una función y su área bajo la curva

- Graficamos una función y sobreamos el área que representa la integral.

In [1]:
```
import numpy as mp
import matplotlib.pyplot as plt
from scipy.integrate import quad
```

In [ ]:
```
def funcion_area(x):
    return np.sin(x) + 0.5 * x

a, b = 0, 5
x = np.linspace(0, 5, 100)
y = funcion_area(x)

resultado_area, _ = quad(funcion_area, a, b)

plt.figure(figsize=(10,5))
plt.plot(x, y, label=r'$f(x) = \sin(x) + 0.5x$', color='blue')
plt.fill_between(x, y, where=((x >= a) & (x <= b)), color='skyblue', alpha=0.5, lab
plt.axhline(0, color='black', linewidth=0.5)
plt.title(f'Integral de f(x) entre [{a}, {b}] = {resultado_area:.4f}')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.legend()
plt.grid()
plt.show()
```



Integral Doble con área en 3D

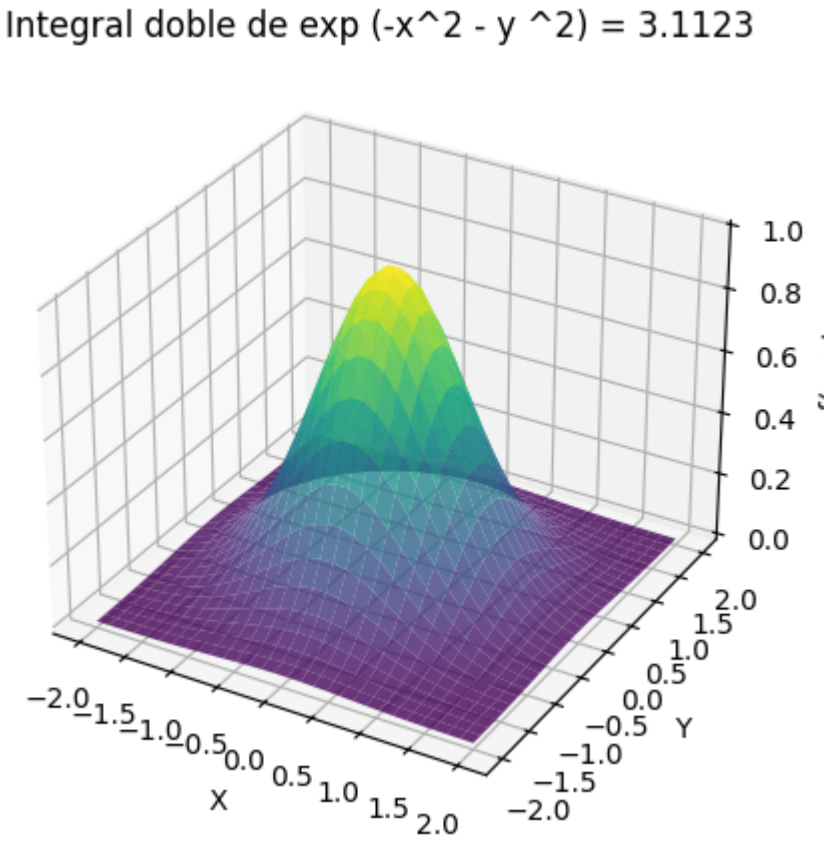
- En este ejemplo graficaremos una integral doble en una superficie tridimensional.

In [7]:
```
from scipy.integrate import dblquad
from mpl_toolkits.mplot3d import Axes3D
import numpy as np
```

In [15]:
```
def funcion_doble(x, y):
    return np.exp(-x**2 - y**2)


x_min, x_max = -2, 2
y_min, y_max = -2, 2
```

```
res, _ = dblquad(funcion_doble, x_min, x_max, lambda x: y_min, lambda x: y_max)

X = np.linspace(x_min, x_max, 30)
Y = np.linspace(y_min, y_max, 30)
X , Y = np.meshgrid(X, Y)
Z = funcion_doble(X, Y)

fig = plt.figure(figsize=(10,5))
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(X, Y, Z, cmap='viridis', alpha=0.8)
ax.set_title(f'Integral doble de exp (-x^2 - y ^2) = {res:.4f}')
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('f(x, y)')
plt.grid()
plt.show
```

Out[15]:
```
<function matplotlib.pyplot.show(close=None, block=None)>
```



Interpolación con SciPy (scipy.interpolate)

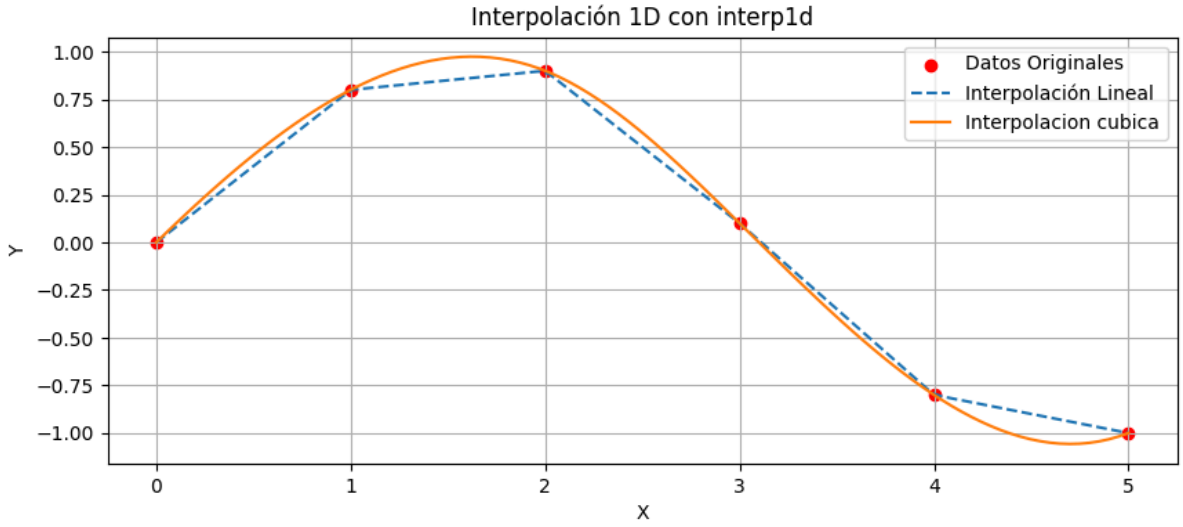
- La interpolacion es una tecnica para estimar valores intermedios entre puntos de datos conocidos.
- SciPy proporciona herramientas para interpolar funciones de una o varias dimensiones.

Interpolación 1D con interp1d

- interp1d permite crear una función interpolante a partir de un conjunto de datos.

In [20]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import interp1d
```

In [23]:
```
x = np.array([0,1,2,3,4,5])
y = np.array([0,0.8,0.9,0.1,-0.8,-1])

interpolacion_linear = interp1d(x, y, kind='linear')
interpolacion_cubica = interp1d(x, y , kind='cubic')

x_new = np.linspace(0, 5, 100)
y_lineal = interpolacion_linear(x_new)
y_cubica = interpolacion_cubica(x_new)

plt.figure(figsize=(10,4))
plt.scatter(x, y, label ='Datos Originales', color='red')
plt.plot(x_new, y_lineal, label='Interpolación Lineal', linestyle='--')
plt.plot(x_new, y_cubica, label='Interpolacion cubica', linestyle = '-')
plt.xlabel('X')
plt.ylabel('Y')
plt.legend()
plt.title('Interpolación 1D con interp1d')
plt.grid(True)
plt.show()
```

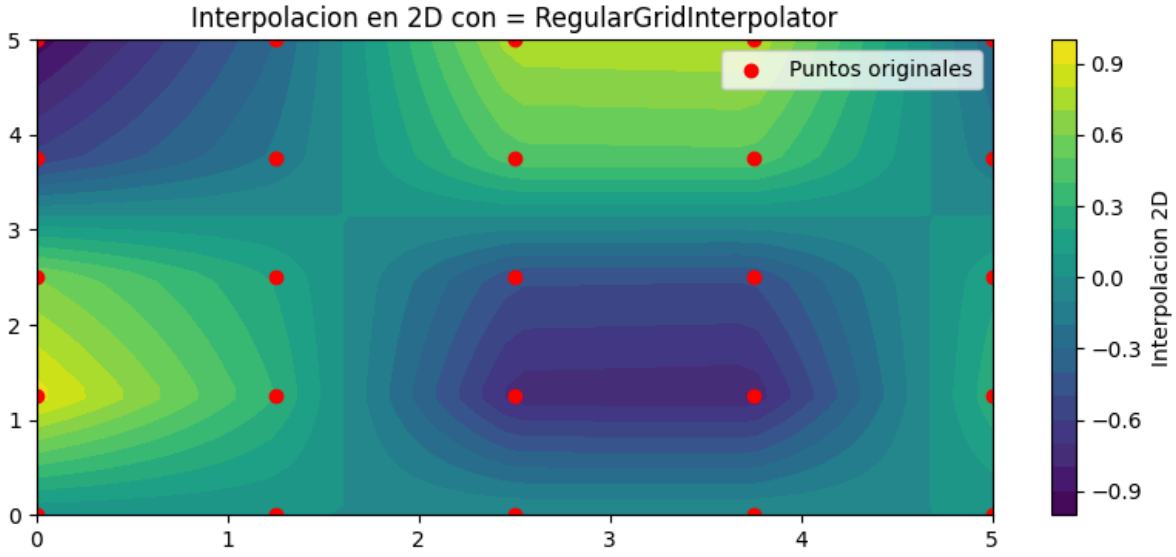


Interpolación con interp2D --> RegularGridInterpolator

- interp2D solia interpolar en 2D, pero ahora (hasta este punto del curso) se recomienda usar RegularGridInterpolator

In [29]:
```
from scipy.interpolate import RegularGridInterpolator

x = np.linspace(0,5,5)
y = np.linspace(0,5,5)
X, Y = np.meshgrid(x, y)
Z = np.sin(X) * np.cos(Y)

funcion_interpolador = RegularGridInterpolator((x, y ), Z)

x_nuevo = np.linspace(0,5,50)
y_nuevo = np.linspace(0,5,50)
X_nuevo, Y_nuevo = np.meshgrid(x_nuevo, y_nuevo)
puntos = np.array([X_nuevo.ravel(), Y_nuevo.ravel()]).T
Z_nuevo = funcion_interpolador(puntos).reshape(50, 50)

plt.figure(figsize=(10,4))
plt.contourf(X_nuevo, Y_nuevo, Z_nuevo, levels=20, cmap='viridis')
plt.colorbar(label='Interpolacion 2D')
```

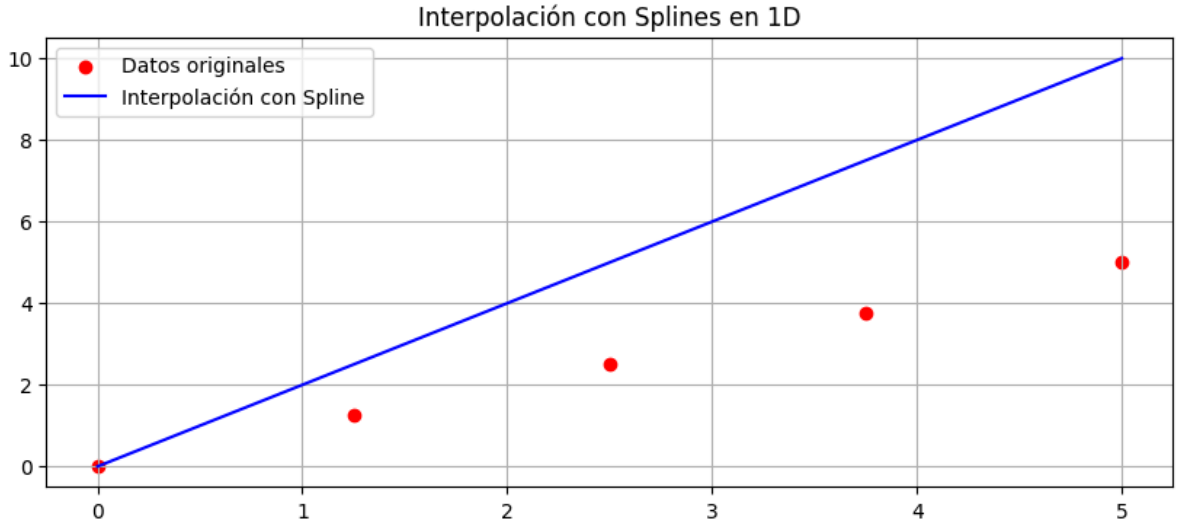
```
plt.scatter(X, Y, color='red', label='Puntos originales')
plt.legend()
plt.title('Interpolacion en 2D con = RegularGridInterpolator')
plt.show()
```



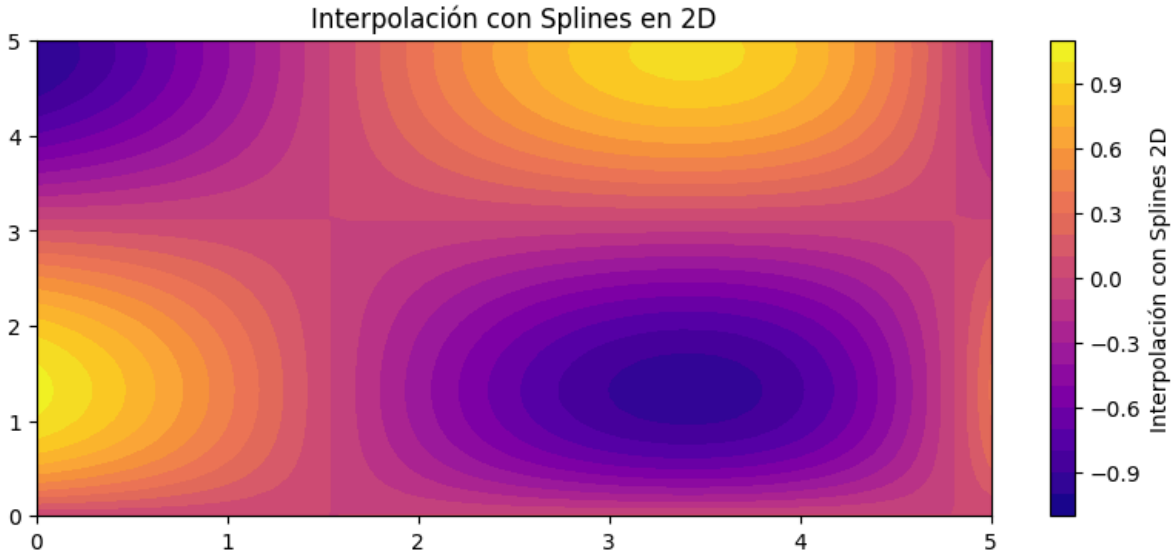
Interpolación con Splines (UnivariatedSpline, BivariateSpline)

- Los splines son funciones suaves para la interpolación

In [32]:
```
from scipy.interpolate import UnivariateSpline

x_in = np.linspace(0,10,10)
y_in =np.sin(x_in)

spline = UnivariateSpline(x, y , s=0.5)

x_n1 = np.linspace(0,10,100)
y_spline = spline(x_n1)

plt.figure(figsize=(10, 4))
plt.scatter(x, y, label="Datos originales", color="red")
plt.plot(x_new, y_spline, label="Interpolación con Spline", color="blue")
plt.legend()
plt.title("Interpolación con Splines en 1D")
plt.grid()
plt.show()
```



In [33]:
```
from scipy.interpolate import bisplrep, bisplev

x = np.linspace(0, 5, 5)
y = np.linspace(0, 5, 5)
X, Y = np.meshgrid(x, y)
Z = np.sin(X) * np.cos(Y)

tck = bisplrep(X.ravel(), Y.ravel(), Z.ravel())

x_n2 = np.linspace(0, 5, 50)
y_n2 = np.linspace(0, 5, 50)
X_n2, Y_n2 = np.meshgrid(x_n2, y_n2)
Z_n2 = bisplev(x_n2, y_n2, tck)

plt.figure(figsize=(10, 4))
plt.contourf(X_n2, Y_n2, Z_n2, levels=20, cmap="plasma")
plt.colorbar(label="Interpolación con Splines 2D")
plt.title("Interpolación con Splines en 2D")
plt.show()
```



Trabajando con Datos de Imágenes (scipy.ndimage)

- SciPy proporciona el módulo scipy.ndimage, una poderosa herramienta para el procesamiento de imágenes multidimensionales.

Cargando transformaciones, necesitamos cargar y visualizar imágenes.

- Antes de realizar transformaciones, necesitamos cargar y visualizar imágenes.
- Para este ejemplo con la ayuda de scikit-image, vamos a a hacer uso de una sus imágenes integradas
- face = data.astronaut() Carga una imagen de ejemplo
- plt. imshow() Muestra la imagen en todo de gris

In [60]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy import ndimage
from skimage import data, color, filters

face = data.astronaut()
```

```
imagen = face

plt.figure(figsize=(6,6))
plt.imshow(imagen, cmap='gray')
plt.title('Imagen Original')
plt.axis('off')
plt.show()
```

Filtro para Suavizar Imágenes

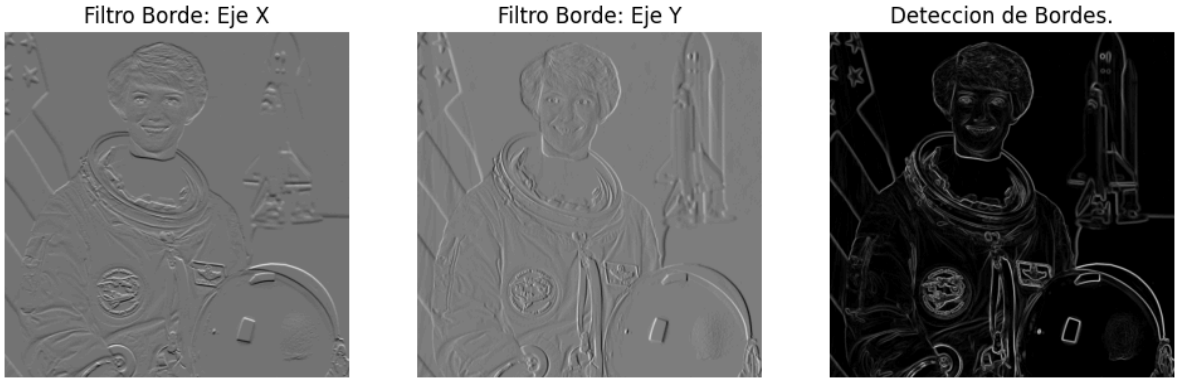
- Podemos aplicar filtros de suavizado para reducir ruido.

In [53]:
```
imagen_filtro = ndimage.gaussian_filter(imagen, sigma=3)

plt.figure(figsize=(6, 6))
plt.imshow(imagen_filtro, cmap="gray")
plt.title("Imagen suavizada con filtro Gaussiano")
plt.axis("off")
plt.show()
```

Detección de Bordes

- Podemos aplicar filtros para resaltar los bordes de la imagen.

In [54]:
```
imagen = color.rgb2gray(data.astronaut()).astype(np.float64)

borde_x = ndimage.sobel(imagen, axis=0)
borde_y = ndimage.sobel(imagen, axis=1)

borde_total = np.hypot(borde_x, borde_y)

fig, axes = plt.subplots(1, 3, figsize=(12,4))

axes[0].imshow(borde_x, cmap='gray')
axes[0].set_title('Filtro Borde: Eje X')
axes[0].axis('off')

axes[1].imshow(borde_y, cmap='gray')
axes[1].set_title('Filtro Borde: Eje Y')
axes[1].axis('off')

axes[2].imshow(borde_total, cmap='gray')
axes[2].set_title('Deteccion de Bordes.')
axes[2].axis('off')

plt.show()
```



Transformaciones Geométricas.

- Podemos modificar la geometría de las imágenes, como rotarlas o escalarlas.

In [55]:
```
imagen_rotada = ndimage.rotate(imagen, 45, reshape=True)

plt.figure(figsize=(6,6))
plt.imshow(imagen_rotada, cmap='gray')
plt.title('Imagen rotada 45°')
plt.axis('off')
plt.show()
```

Escalar una imagen

In [56]:
```
imagen_aumentada = ndimage.zoom(imagen, 0.5)

plt.figure(figsize=(6,6))
```

```
plt.imshow(imagen_aumentada, cmap='gray')
plt.title('Imagen Escalada al 50%')
plt.axis('off')
plt.show()
```

Transformaciones Morfológicas.

- Podemos modificar la forma de las imágenes utilizando operaciones como la erosión o dilatación

In [ ]:
```
from scipy.ndimage import binary_erosion, binary_dilation

imagen_gris = color.rgb2gray(data.astronaut())


umbral = filters.threshold_otsu(imagen_gris)
imagen_binaria = imagen_gris > umbral

estructura = np.ones((5, 5))

imagen_erosion = binary_erosion(imagen_binaria)
imagen_dilatada = binary_dilation(imagen_binaria)

fig, axes = plt.subplots(1, 3, figsize=(12,4))

axes[0].imshow(imagen_binaria, cmap='gray')
axes[0].set_title('Imagen Binaria')
axes[0].axis('off')

axes[1].imshow(imagen_erosion, cmap='gray')
axes[1].set_title('Imagen Erosionada')
```

```
axes[1].axis('off')

axes[2].imshow(imagen_dilatada, cmap='gray')
axes[2].set_title('Imagen Dilatada')
axes[2].axis('off')

plt.show()
```

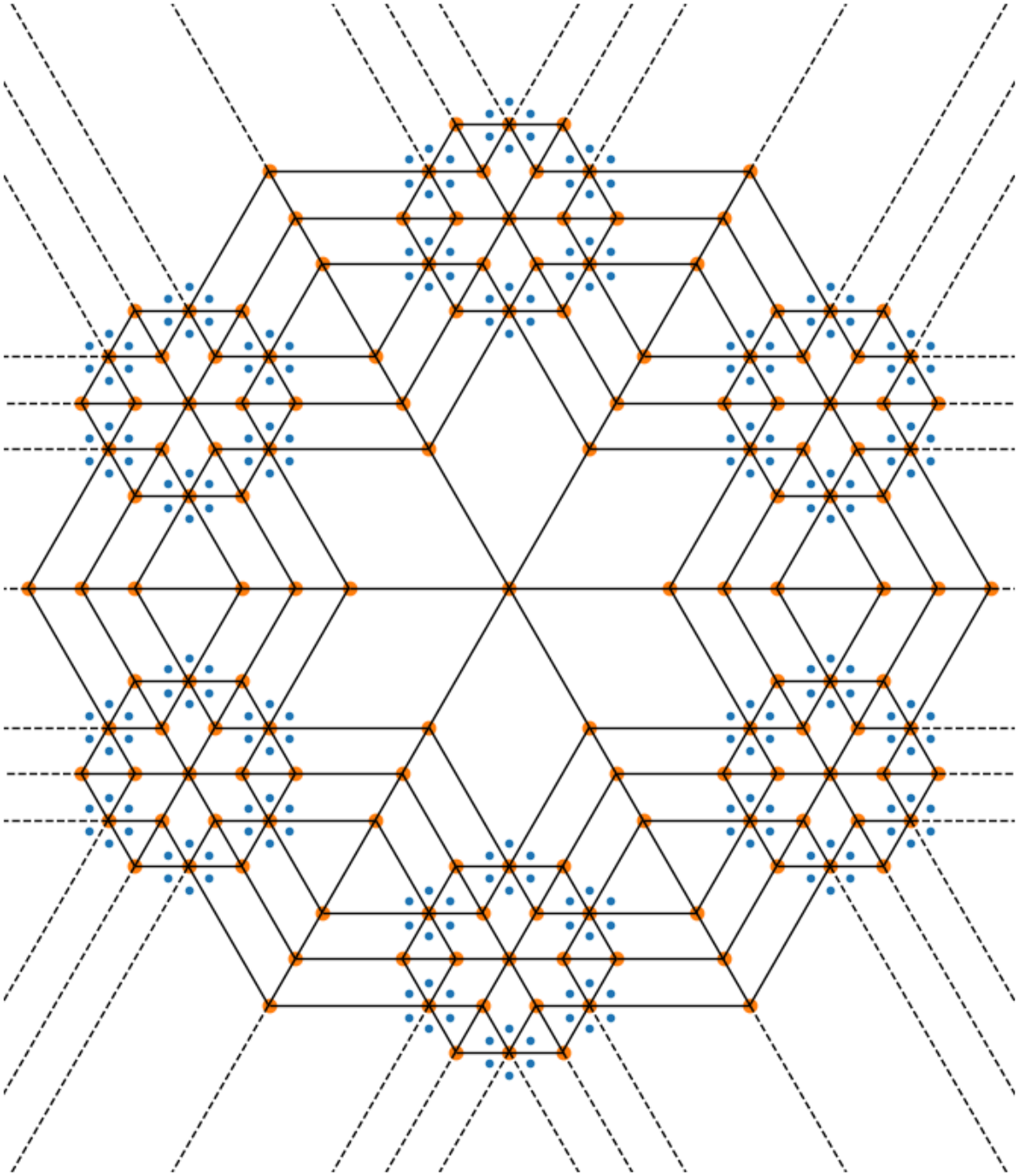
Especial Mandala

In [62]:
```
import numpy as np
from scipy import spatial
import matplotlib.pyplot as plt
```

In [68]:
```
def mandala(n_iter, n_puntos, radio):
    fig = plt.figure(figsize=(10,10))
    ax = fig.add_subplot(111)
    ax.set_axis_off()
    ax.set_aspect('equal', adjustable='box')
    angles = np.linspace(0, 2*np.pi * (1 -1/n_puntos), num=n_puntos) + np.pi/2
    xy = np.array([[0,0]])
    for k in range(n_iter):
        t1 = np.array([])
        t2 = np.array([])
        for i in range(xy.shape[0]):
            t1 = np.append(t1, xy[i ,0] + radio**k * np.cos(angles))
            t2 = np.append(t2, xy[i ,1] + radio**k * np.sin(angles))

        xy = np.column_stack((t1, t2))

    spatial.voronoi_plot_2d(spatial.Voronoi(xy), ax=ax)
    return fig
```

In [69]:
```
n_iter = 3
n_puntos = 6
radio = 4
```

In [70]:
```
fig = mandala(n_iter, n_puntos, radio)
plt.show()
```

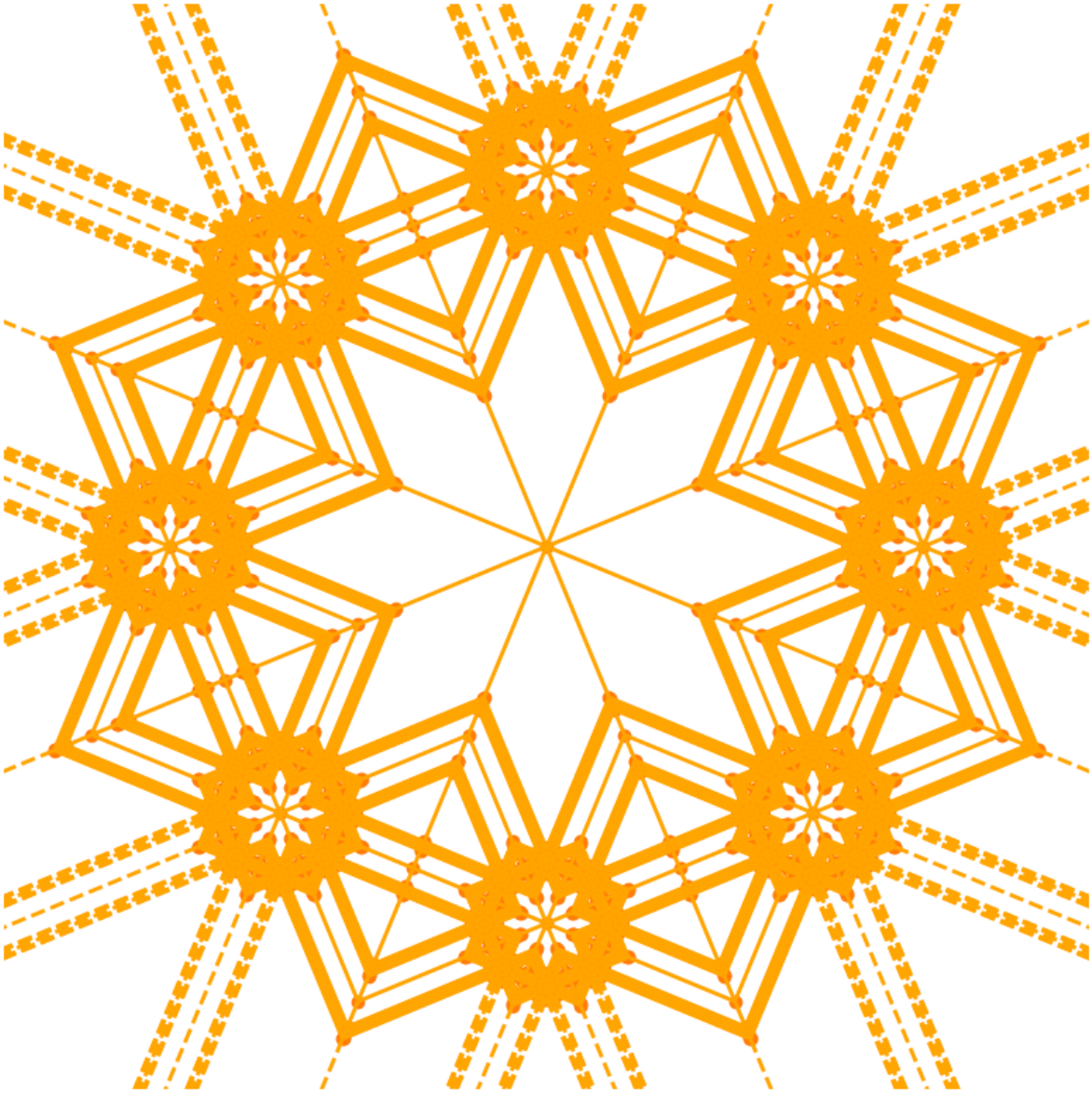


In [71]:

```
def mandala(n_iter, n_puntos, radio):
    fig = plt.figure(figsize=(10,10))
    ax = fig.add_subplot(111)
    ax.set_axis_off()
    ax.set_aspect('equal', adjustable='box')
    angles = np.linspace(0, 2*np.pi * (1 -1/n_puntos), num=n_puntos) + np.pi/2
    xy = np.array([[0,0]])
    for k in range(n_iter):
        t1 = np.array([])
        t2 = np.array([])
        for i in range(xy.shape[0]):
            t1 = np.append(t1, xy[i ,0] + radio**k * np.cos(angles))
            t2 = np.append(t2, xy[i ,1] + radio**k * np.sin(angles))

        xy = np.column_stack((t1, t2))

    spatial.voronoi_plot_2d(spatial.Voronoi(xy), ax=ax, line_width=2, line_colors='
    return fig
```

In [72]:
```
n_iter = 5
n_puntos = 8
radio = 6
```

In [73]:
```
fig = mandala(n_iter, n_puntos, radio)
plt.show()
```



Por: Eduardo Soto ING de Software