Imports

In [1]:
```
# These are the standard imports for CS 111.
# This list may change as the quarter goes on.
import os
import time
import math
import numpy as np
import numpy.linalg as npla
import scipy
from scipy import linalg as spla
import scipy.sparse
import scipy.sparse.linalg
from scipy import integrate
import matplotlib.pyplot as plt
from matplotlib import cm
import networkx as nx
from mpl_toolkits.mplot3d import axes3d
import cs111
%matplotlib ipympl
%matplotlib inline
np.set_printoptions(precision = 4)
```

Introduction

Model: law or formula that represents a given situation
Discretization: splitting a continuous variable into a discrete amount of time to achieve a solution (using a linear system of equations)

Fundamental Form of Linear System of Equations: Ax = b

The Geometry of Ax = b

- Linear system of equations with n unknowns and n equations
- **Row picture:** one equation at a time
- **Column picture:** linear combination of column vectors
- **Matrix picture:** coefficients of matrix

Example:

$$\begin{pmatrix} 2x - y = 0 \\ x + 2y = 3 \end{pmatrix}$$

Matrix Picture:

$$\begin{pmatrix} 2 & -1 \\ -1 & 2 \end{pmatrix} * \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 \\ 3 \end{pmatrix}$$

Row Picture:
Find intersection of the equations:

$$\begin{pmatrix} 2x - y = 0 \\ x + 2y = 3 \end{pmatrix}$$

Column Picture:
Find the linear combination of the columns:

$$x * \begin{pmatrix} 2 \\ -1 \end{pmatrix} + y * \begin{pmatrix} -1 \\ 2 \end{pmatrix} = \begin{pmatrix} 0 \\ 3 \end{pmatrix}$$

Matrices

Dense Matrix: matrix with lots of values
Sparse Matrix: matrix with many null values

In [2]:
```
# Matrix
S = np.array([[2, -1], [-1, 2]])
print("S:", S, sep="\n")

# Vector
b = np.array([0, 3]) #doesn't matter if column or row vector
print("b:", b)

# Solve Ax = b
x = npla.solve(S, b) #solves (dense) linear system of equations
print("x:", x)

#Row and Column Picture
v = np.arange(2)
print("Linear Combination of Columns:", S @ v, sep="\n")
print("Linear Combination of Rows:", v @ S, sep="\n")
print("Transpose:", S.T, sep="\n")
print("Transpose:", v.T, sep="\n")

#Equality Check
print("Element Wise Equality:", S.T.T == S, sep="\n")
print("Overall Equality:", (S.T.T == S).all())

# Matrix Operations
print("Matrix Multiplication", S @ x, sep="\n")
print("Matrix Addition", S + 3, sep="\n")
print("Matrix Addition", S + S, sep="\n")
print("Element Wise Multiplication", S * S, sep="\n")
print("Scalar Multiplication", S / 2, sep="\n")
print("Element Wise Sin", np.sin(S), sep="\n")
```

```
S:
[[ 2 -1]
 [-1  2]]
b: [0 3]
x: [1. 2.]
Linear Combination of Columns:
[-1  2]
Linear Combination of Rows:
[-1  2]
Transpose:
[[ 2 -1]
 [-1  2]]
Transpose:
[0 1]
Element Wise Equality:
[[ True  True]
 [ True  True]]
Overall Equality: True
Matrix Multiplication
[0. 3.]
Matrix Addition
[[5 2]
 [2 5]]
Matrix Addition
[[ 4 -2]
 [-2  4]]
Element Wise Multiplication
[[4 1]
 [1 4]]
Scalar Multiplication
[[ 1.  -0.5]
 [-0.5  1. ]]
```

```
Element Wise Sin
[[ 0.9093 -0.8415]
 [-0.8415  0.9093]]
```

In [3]:
```
#Numpy Array Range vs Python List Range
R = np.arange(16)
print("Numpy Array:", R)
print("Python List:", list(range(16)))

# Reshape
print("Reshape R:", R.reshape(4, 4))
# R.reshape? #refer to documentation

# Random Matrices
A = np.random.random((4, 4))
A = np.round(100 * A)
print("A:", A, sep="\n")

#One Dimensional Array Accessing
print("Array Indexing:", R[5])
print("Array Subset:", R[4:8])
print("Array Prefix:", R[:8])
print("Array Subfix:", R[6:])
print("Array Subset with Step Size:", R[2:12:2])
print("Reverse Array:", R[::-1])

#Two Dimensional Array Accessing
print("Array Indexing:", A[2,3])
print("Array Subset:", A[2,:])
print("Array Subset Reverse:", A[2,::-1])
print("Array Subset Row and Col:", A[::-1,::-1], sep="\n")

#Array Modification
A[2, 0] = 3
A[[0, 1], :] = A[[1, 0], :] #swap rows 0 and 1
print("A:", A, sep="\n")
```

```
Numpy Array: [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15]
Python List: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
Reshape R: [[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
A:
[[87. 11. 92. 76.]
 [69. 31. 88. 81.]
 [ 7. 86. 81. 87.]
 [21. 64. 67. 52.]]
Array Indexing: 5
Array Subset: [4 5 6 7]
Array Prefix: [0 1 2 3 4 5 6 7]
Array Subfix: [ 6  7  8  9 10 11 12 13 14 15]
Array Subset with Step Size: [ 2  4  6  8 10]
Reverse Array: [15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0]
Array Indexing: 87.0
Array Subset: [ 7. 86. 81. 87.]
Array Subset Reverse: [87. 81. 86.  7.]
Array Subset Row and Col:
[[52. 67. 64. 21.]
 [87. 81. 86.  7.]
 [81. 88. 31. 69.]
 [76. 92. 11. 87.]]
A:
[[69. 31. 88. 81.]
 [87. 11. 92. 76.]
 [ 3. 86. 81. 87.]
 [21. 64. 67. 52.]]
```

Matrix Arithmetic

In [4]:
```
A = np.arange(9).reshape((3, 3))
print("A:", A, sep="\n")

B = np.round(10 * np.random.random((3, 3)))
print("B:", B, sep="\n")

C = A + B #O(n^2) where size of the matrix is n x n
print("C:", C, sep="\n")

C = A * B #O(n^2)
print("C:", C, sep="\n")

C = A @ B #O(n^3) because n^2 elements applying dot product (n operations)
print("C:", C, sep="\n")

x = np.array([3, 1, 4])
y = A @ x #O(n^2) because n dot products
print("y:", y)

a = npla.norm(x) #O(n) access one element each time
print("x:", x, "norm(x):", a)
```

```
A:
[[0 1 2]
 [3 4 5]
 [6 7 8]]
B:
[[ 7.  7.  9.]
 [ 9. 10.  7.]
 [ 3.  5.  3.]]
C:
[[ 7.  8. 11.]
 [12. 14. 12.]
 [ 9. 12. 11.]]
C:
[[ 0.  7. 18.]
 [27. 40. 35.]
 [18. 35. 24.]]
C:
[[ 15.  20.  13.]
 [ 72.  86.  70.]
 [129. 152. 127.]]
y: [ 9 33 57]
x: [3 1 4] norm(x): 5.0990195135927845
```

Interesting Matrices

In [5]:
```
print("Zero Matrix:", np.zeros((4, 4)), sep="\n")
print("One Matrix:", np.ones((4, 4)), sep="\n")
I = np.eye(4)
print("Identity Matrix:", I, sep="\n")
D = np.diag([2, 1, .5, 0])
print("Diagonal Matrix:", D, sep="\n")
```

```
Zero Matrix:
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
One Matrix:
[[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]]
Identity Matrix:
[[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]]
Diagonal Matrix:
[[2.  0.  0.  0. ]
 [0.  1.  0.  0. ]
 [0.  0.  0.5 0. ]
 [0.  0.  0.  0. ]]
```

In [6]:
```
A = np.arange(16).reshape((4, 4))
print("D @ A:", D @ A, sep="\n") #scaling rows of A
print("A @ D:", A @ D, sep="\n") #scaling columns of A
```

```
D @ A:
[[0.  2.  4.  6. ]
 [4.  5.  6.  7. ]
 [4.  4.5 5.  5.5]
 [0.  0.  0.  0. ]]
A @ D:
[[ 0.  1.  1.  0.]
 [ 8.  5.  3.  0.]
 [16.  9.  5.  0.]
 [24. 13.  7.  0.]]
```

Lower Triangular Matrix

In [7]:
```
L = np.tril(A) #sets upper triangle (exc. diagonal) to 0
L
```

Out[7]:
```
array([[ 0,  0,  0,  0],
       [ 4,  5,  0,  0],
       [ 8,  9, 10,  0],
       [12, 13, 14, 15]])
```

In [8]:
```
#Unit Lower Triangular (diagonal = 1)
L = np.array([[1, 0, 0, 0], [2, 2, 0, 0], [2, 0, 1, 0], [-1, 2, 1, 1]])
L
```

Out[8]:
```
array([[ 1,  0,  0,  0],
       [ 2,  2,  0,  0],
       [ 2,  0,  1,  0],
       [-1,  2,  1,  1]])
```

In [9]:
```
b = np.array([2, 6, 3, 1])
y = npla.solve(L, b) #inefficient
y
```

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

In [10]:
```
y = cs111.Lsolve(L, b) #use lower triangular property for efficiency
y
```

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

In [11]:
```
L @ y
```

Out[11]:
```
array([2., 6., 3., 1.])
```

In [12]:
```
def Lsolve(L, b):
    #create new list for transformation
    solution = b.copy()

    for i in range(len(L)):
        #matrix cannot be singular
        assert L[i][i] != 0

        #transform to unit triangular
        solution[i] /= L[i][i]

        #transform new list
        solution[i+1:] -= solution[i] * L[i+1:,i]

    #return final solution
    return solution
```

In [13]:
```
Lsolve(L, b)
```

Out[13]:
```
array([ 2,  1, -1,  2])
```

In [14]:
```
print(L, b)
```

```
[[ 1  0  0  0]
 [ 2  2  0  0]
 [ 2  0  1  0]
 [-1  2  1  1]] [2 6 3 1]
```

In [15]:
```
def Usolve(U, b):
    '''solve linear system given upper triangular matrix (U) and result vector (b)'''
    #constraints on U
    row, col = U.shape
    assert row == col #must be square
    assert np.all(np.triu(U) == U) #must be upper triangular

    #constraints on b
    assert b.ndim == 1 #must be 1 dimensional
    assert b.shape[0] == row #must be same size as U

    #create new list for transformation
    solution = b.astype(np.float64).copy()

    for i in reversed(range(len(U))):
        #matrix cannot be singular
        assert U[i][i] != 0

        #transform to unit triangular
        solution[i] /= U[i][i]

        #transform new list
        solution[:i] -= solution[i] * U[:i,i]

    #return final solution
    return solution
```

Linear Combinations

Right-multiplication: linear combination of columns

$$\begin{pmatrix} a_0 & a_1 & a_2 \\ a_3 & a_4 & a_5 \\ a_6 & a_7 & a_8 \end{pmatrix} * \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = x_0 * \begin{pmatrix} a_0 \\ a_3 \\ a_6 \end{pmatrix} + x_1 * \begin{pmatrix} a_1 \\ a_4 \\ a_7 \end{pmatrix} + x_2 * \begin{pmatrix} a_2 \\ a_5 \\ a_8 \end{pmatrix}$$

Left-multiplication: linear combination of rows

$$\begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} * \begin{pmatrix} a_0 & a_1 & a_2 \\ a_3 & a_4 & a_5 \\ a_6 & a_7 & a_8 \end{pmatrix} = x_0 * \begin{pmatrix} a_0 & a_1 & a_2 \end{pmatrix} + x_1 * \begin{pmatrix} a_3 & a_4 & a_5 \end{pmatrix} + x_2 * \begin{pmatrix} a_6 & a_7 & a_8 \end{pmatrix}$$

Permutations

n-Permutation: list of the numbers 0 through n-1 in some order

Example: $\begin{pmatrix} 3 & 4 & 0 & 1 & 2 \end{pmatrix}$ is a 5-permutation

nxn-Permuation Matrix: matrix of 0s and 1s containing exactly one 1 in each row and column

Example:

$$\begin{pmatrix} 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

Fact: Every permutation list corresponds to a permutation matrix.

Example: Given vector v = $\begin{pmatrix} 3.1 & 4.1 & 5.9 & 2.6 & 5.3 \end{pmatrix}$ and the above permutation matrix, we can determine a permutation of v as follows:

$$\begin{pmatrix} 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix} * \begin{pmatrix} 3.1 \\ 4.1 \\ 5.9 \\ 2.6 \\ 5.3 \end{pmatrix} = \begin{pmatrix} 2.6 \\ 3.1 \\ 5.3 \\ 4.1 \\ 5.9 \end{pmatrix}$$

Theorem: The transpose of a permuation matrix is its inverse.

Proof: The ith row of P transpose is the ith column of P. The dot product of the rows is 1 if i = j (since 1 in same position) or 0 if i != j (since 1 is in different position).

In [16]:
```
v = np.array([3.1, 4.1, 5.9, 2.6, 5.3])

#permutation
p = [3, 0, 4, 1, 2]
vp = v[p]
print("v[p]:", vp)

#permutation matrix
P = np.array([[0, 0, 0, 1, 0], [1, 0, 0, 0, 0], [0, 0, 0, 0, 1], [0, 1, 0, 0, 0], [0,
print("P @ v:", P @ v)
```

```
v[p]: [2.6 3.1 5.3 4.1 5.9]
P @ v: [2.6 3.1 5.3 4.1 5.9]
```

In [17]:
```
pinv = [1, 3, 4, 0, 2]
print("v[pinv]:", v[pinv])
print("P.T @ v:", P.T @ v)

#inverse permutation
print("vp[pinv]", vp[pinv])
print("v[p][pinv]", v[p][pinv]) #permutes left to right
print("P.T @ P @ v:", P.T @ P @ v)

#tranpose of a permutation matrix is its inverse
print("P.T @ P", P.T @ P, sep="\n")
print("P @ P.T", P @ P.T, sep="\n")
```

```
v[pinv]: [4.1 2.6 5.3 3.1 5.9]
P.T @ v: [4.1 2.6 5.3 3.1 5.9]
vp[pinv] [3.1 4.1 5.9 2.6 5.3]
v[p][pinv] [3.1 4.1 5.9 2.6 5.3]
P.T @ P @ v: [3.1 4.1 5.9 2.6 5.3]
P.T @ P
[[1 0 0 0 0]
 [0 1 0 0 0]
 [0 0 1 0 0]
 [0 0 0 1 0]
 [0 0 0 0 1]]
P @ P.T
[[1 0 0 0 0]
 [0 1 0 0 0]
 [0 0 1 0 0]
 [0 0 0 1 0]
 [0 0 0 0 1]]
```

In [18]:
```
#permute a matrix
A = (20 * np.random.random((5, 5))).round()
print("A:", A, sep="\n")

#permute the rows
print("A[p,:]", A[p,:], sep="\n") #using a list
print("P @ A", P @ A, sep="\n") #using a permutation matrix

#permute the columns
print("A[:,p]", A[:,p], sep="\n") #using a list
print("A @ P.T", A @ P.T, sep="\n") #using a permutation matrix
#note that we need to use transpose for left to right order
```

```
A:
[[ 8.  4.  8. 18.  2.]
 [ 7.  7.  6.  2.  1.]
 [ 1.  7.  2. 15. 14.]
 [19.  5.  8.  4. 12.]
 [ 9.  3. 17. 12. 15.]]
A[p,:]
[[19.  5.  8.  4. 12.]
 [ 8.  4.  8. 18.  2.]
 [ 9.  3. 17. 12. 15.]
 [ 7.  7.  6.  2.  1.]
 [ 1.  7.  2. 15. 14.]]
P @ A
[[19.  5.  8.  4. 12.]
 [ 8.  4.  8. 18.  2.]
 [ 9.  3. 17. 12. 15.]
 [ 7.  7.  6.  2.  1.]
 [ 1.  7.  2. 15. 14.]]
A[:,p]
[[18.  8.  2.  4.  8.]
 [ 2.  7.  1.  7.  6.]
 [15.  1. 14.  7.  2.]
 [ 4. 19. 12.  5.  8.]
 [12.  9. 15.  3. 17.]]
A @ P.T
[[18.  8.  2.  4.  8.]
 [ 2.  7.  1.  7.  6.]
 [15.  1. 14.  7.  2.]
 [ 4. 19. 12.  5.  8.]
 [12.  9. 15.  3. 17.]]
```

Temperature Problem

5-point stensil: temperature at a point depends on its four adjacent neightbors
Poisson Equation: $x_i = 0.25(x_{i-k} + x_{i-1} + x_{i+1} + x_{i+k})$
We can rearrange to be a linear system: $-x_{i-k} - x_{i-1} + 4x_i - x_{i+1} - x_{i+k} = 0$
When next to a wall, we can substitute for the value of the wall and rearrange for the constant to be on the right side

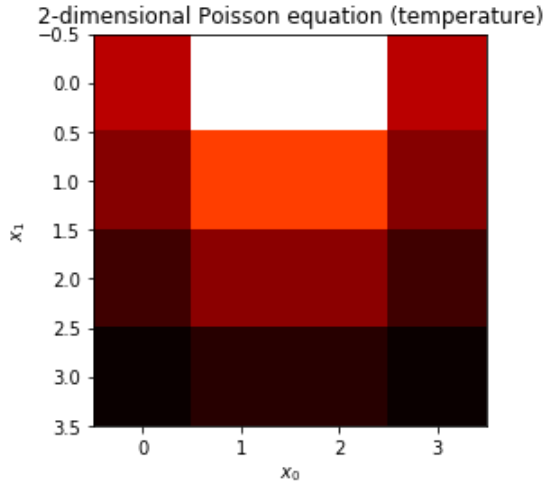
A is a sparse matrix containing at most 5 nonzero values.

In [19]:
```
#Small Example: k = 4, n = 16 unknowns
A = cs111.make_A_small()
print("A:", A, sep="\n")

b = cs111.make_b(4, top=[32, 212, 212, 32], bottom=32, left=32, right=32)
print("b:", b, sep="\n")

x = npla.solve(A, b)
print("x:", x, sep="\n")

T = x.reshape(4, 4)
print("T:", T, sep="\n")
```

```
A:
[[ 4. -1.  0.  0. -1.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [-1.  4. -1.  0.  0. -1.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0. -1.  4. -1.  0.  0. -1.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0. -1.  4.  0.  0.  0. -1.  0.  0.  0.  0.  0.  0.  0.  0.]
 [-1.  0.  0.  0.  4. -1.  0.  0. -1.  0.  0.  0.  0.  0.  0.  0.]
 [ 0. -1.  0.  0. -1.  4. -1.  0.  0. -1.  0.  0.  0.  0.  0.  0.]
 [ 0.  0. -1.  0.  0. -1.  4. -1.  0.  0. -1.  0.  0.  0.  0.  0.]
 [ 0.  0.  0. -1.  0.  0. -1.  4.  0.  0.  0. -1.  0.  0.  0.  0.]
 [ 0.  0.  0.  0. -1.  0.  0.  0.  4. -1.  0.  0. -1.  0.  0.  0.]
 [ 0.  0.  0.  0.  0. -1.  0.  0. -1.  4. -1.  0.  0. -1.  0.  0.]
 [ 0.  0.  0.  0.  0.  0. -1.  0.  0. -1.  4. -1.  0.  0. -1.  0.]
 [ 0.  0.  0.  0.  0.  0.  0. -1.  0.  0. -1.  4.  0.  0.  0. -1.]
 [ 0.  0.  0.  0.  0.  0.  0.  0. -1.  0.  0.  0.  4. -1.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0. -1.  0.  0. -1.  4. -1.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0. -1.  0.  0. -1.  4. -1.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0. -1.  0.  0. -1.  4.]]
b:
[ 64. 212. 212.  64.  32.   0.   0.  32.  32.   0.   0.  32.  64.  32.
  32.  64.]
x:
[ 57.2273 113.8182 113.8182  57.2273  51.0909  72.2273  72.2273  51.0909
  42.9091  51.7727  51.7727  42.9091  36.7727  40.1818  40.1818  36.7727]
T:
[[ 57.2273 113.8182 113.8182  57.2273]
 [ 51.0909  72.2273  72.2273  51.0909]
 [ 42.9091  51.7727  51.7727  42.9091]
 [ 36.7727  40.1818  40.1818  36.7727]]
```

In [20]:
```
plt.figure()
plt.imshow(T, cmap=cm.hot)
plt.xlabel(r'$x_0$')
plt.ylabel(r'$x_1$')
plt.title(r'2-dimensional Poisson equation (temperature)')
```

Out[20]:
```
Text(0.5, 1.0, '2-dimensional Poisson equation (temperature)')
```



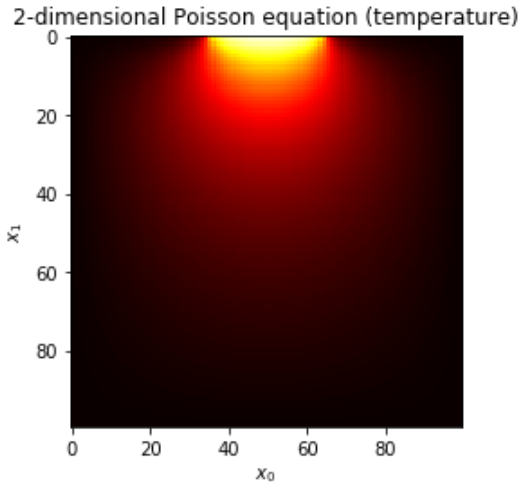
In [21]:
```
#Large Example: k = 100, n = 10,000 unknowns (using sparse data structs)
k=100
A = cs111.make_A(k)
print("A:", A, sep="\n")

b = cs111.make_b(k, top=cs111.radiator(k, temperature=212, default=32), bottom=32, le
print("b:", b, sep="\n")

x = scipy.sparse.linalg.spsolve(A, b)
print("x:", x, sep="\n")

T = x.reshape(100, 100)
print("T:", T, sep="\n")
```

```
A:
  (0, 0)        4.0
  (0, 1)        -1.0
  (0, 100)      -1.0
  (1, 0)        -1.0
  (1, 1)        4.0
  (1, 2)        -1.0
  (1, 101)      -1.0
  (2, 1)        -1.0
  (2, 2)        4.0
  (2, 3)        -1.0
  (2, 102)      -1.0
  (3, 2)        -1.0
  (3, 3)        4.0
  (3, 4)        -1.0
  (3, 103)      -1.0
  (4, 3)        -1.0
  (4, 4)        4.0
  (4, 5)        -1.0
  (4, 104)      -1.0
  (5, 4)        -1.0
  (5, 5)        4.0
  (5, 6)        -1.0
  (5, 105)      -1.0
  (6, 5)        -1.0
  (6, 6)        4.0
  :     :
  (9993, 9993)  4.0
  (9993, 9994)  -1.0
  (9994, 9894)  -1.0
  (9994, 9993)  -1.0
  (9994, 9994)  4.0
  (9994, 9995)  -1.0
  (9995, 9895)  -1.0
  (9995, 9994)  -1.0
  (9995, 9995)  4.0
  (9995, 9996)  -1.0
  (9996, 9896)  -1.0
  (9996, 9995)  -1.0
  (9996, 9996)  4.0
  (9996, 9997)  -1.0
  (9997, 9897)  -1.0
  (9997, 9996)  -1.0
  (9997, 9997)  4.0
  (9997, 9998)  -1.0
  (9998, 9898)  -1.0
  (9998, 9997)  -1.0
  (9998, 9998)  4.0
  (9998, 9999)  -1.0
```

```
  (9999, 9899)  -1.0
  (9999, 9998)  -1.0
  (9999, 9999)  4.0
b:
[64. 32. 32. ... 32. 32. 64.]
x:
[32.0622 32.1247 32.1881 ... 32.0256 32.0171 32.0085]
T:
[[32.0622 32.1247 32.1881 ... 32.1881 32.1247 32.0622]
 [32.1239 32.2487 32.375  ... 32.375  32.2487 32.1239]
 [32.1849 32.371  32.5594 ... 32.5594 32.371  32.1849]
 ...
 [32.0256 32.0512 32.0768 ... 32.0768 32.0512 32.0256]
 [32.0171 32.0341 32.0512 ... 32.0512 32.0341 32.0171]
 [32.0085 32.0171 32.0256 ... 32.0256 32.0171 32.0085]]
```

In [22]:
```
plt.figure()
plt.imshow(T, cmap=cm.hot)
plt.xlabel(r'$x_0$')
plt.ylabel(r'$x_1$')
plt.title(r'2-dimensional Poisson equation (temperature)')
```

Out[22]:
```
Text(0.5, 1.0, '2-dimensional Poisson equation (temperature)')
```



Gaussian Elimination

Matrix Factorization: decomposing a matrix A into two simpler matrices, R and S, such that $A = RS$.

Want to solve Ax = b => (RS)x = b => R(Sx) = b.
Define Sx = y. Then, Ry = b.
Solve for y in Ry = b. Then, solve for x in Sx = y.

Gaussian Elimination: row-reduction by subtracting rows from other rows to form a triangular matrix (O(n^3))

Example:

$$\begin{pmatrix} 2 & 7 & 1 & 8 \\ 1 & 5.5 & 8.5 & 5 \\ 0 & 1 & 12 & 2.5 \\ -1 & -4.5 & -4.5 & 3.5 \end{pmatrix} * \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 18 \\ 20 \\ 15.5 \\ 16.5 \end{pmatrix}$$

First Transformation:

$$M = \begin{pmatrix} . & . & . & . \\ 0.5 & . & . & . \\ 0 & . & . & . \\ -0.5 & . & . & . \end{pmatrix}$$

$$A = \begin{pmatrix} 2 & 7 & 1 & 8 \\ 0 & 2 & 8 & 1 \\ 0 & 1 & 12 & 2.5 \\ 0 & -1 & -4 & 7.5 \end{pmatrix}$$

Second Transformation:

$$M = \begin{pmatrix} . & . & . & . \\ 0.5 & . & . & . \\ 0 & 0.5 & . & . \\ -0.5 & -0.5 & . & . \end{pmatrix}$$

$$A = \begin{pmatrix} 2 & 7 & 1 & 8 \\ 0 & 2 & 8 & 1 \\ 0 & 0 & 8 & 2 \\ 0 & 0 & 0 & 8 \end{pmatrix}$$

Third Transformation:

$$M = \begin{pmatrix} . & . & . & . \\ 0.5 & . & . & . \\ 0 & 0.5 & . & . \\ -0.5 & -0.5 & 0 & . \end{pmatrix}$$

$$A = \begin{pmatrix} 2 & 7 & 1 & 8 \\ 0 & 2 & 8 & 1 \\ 0 & 0 & 8 & 2 \\ 0 & 0 & 0 & 8 \end{pmatrix}$$

Final Result:

$$L = M = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0.5 & 1 & 0 & 0 \\ 0 & 0.5 & 1 & 0 \\ -0.5 & -0.5 & 0 & 1 \end{pmatrix}$$

$$U = A = \begin{pmatrix} 2 & 7 & 1 & 8 \\ 0 & 2 & 8 & 1 \\ 0 & 0 & 8 & 2 \\ 0 & 0 & 0 & 8 \end{pmatrix}$$

where the pivots are the diagonal elements of A.

In [23]:
```
L = np.array([[1, 0, 0, 0], [.5, 1, 0, 0], [0, .5, 1, 0], [-.5, -.5, 0, 1]])
print("L: ", L, sep="\n")

b = np.random.rand(4)
print("b: ", b, sep="\n")

y = cs111.Lsolve(L, b)
print("y: ", y, sep="\n")

#Residuals:
print("b - Ly:", b - L @ y, sep="\n")
```

```
L:
[[ 1.   0.   0.   0. ]
 [ 0.5  1.   0.   0. ]
 [ 0.   0.5  1.   0. ]
 [-0.5 -0.5  0.   1. ]]
b:
[0.8136 0.3451 0.5435 0.133 ]
y:
[ 0.8136 -0.0617  0.5744  0.509 ]
b - Ly:
[ 0.0000e+00  0.0000e+00  1.1102e-16 -1.1102e-16]
```

In [24]:
```
U = np.array([[2, 7, 1, 8], [0, 2, 8, 1], [0, 0, 8, 2], [0, 0, 0, 8]])
print("U: ", U, sep="\n")

b = np.random.rand(4)
print("b: ", b, sep="\n")

x = Usolve(U, b)
print("x: ", x, sep="\n")

#Residuals:
print("b - Ux:", b - U @ x, sep="\n")
```

```
U:
[[2 7 1 8]
 [0 2 8 1]
 [0 0 8 2]
 [0 0 0 8]]
b:
[0.3702 0.1843 0.3675 0.6588]
x:
[ 0.0195 -0.0504  0.0254  0.0823]
b - Ux:
[0. 0. 0. 0.]
```

In [25]:
```
A = L @ U
y = cs111.Lsolve(L, b)
x = Usolve(U, y)
print("y:", y, sep="\n")
print("x:", x, sep="\n")

print("A @ x:", A @ x, sep="\n")
print("b:", b, sep="\n")

#residuals occur due to roundoff errors from finite bits
print("b - A @ x:", b - A @ x, sep="\n")
```

```
y:
[ 3.7020e-01 -7.9778e-04  3.6791e-01  8.4349e-01]
x:
[ 0.2143 -0.1316  0.0196  0.1054]
A @ x:
[0.3702 0.1843 0.3675 0.6588]
b:
[0.3702 0.1843 0.3675 0.6588]
b - A @ x:
[ 0.0000e+00 -1.1102e-16  0.0000e+00  0.0000e+00]
```

LU Factorization

Pivoting: swapping rows to prevent pivot being 0 by constructing a permuation

In [26]:
```
A = np.random.rand(6, 6)
print("A:", A, sep="\n")

b = np.random.rand(6)
print("b:", b, sep="\n")

#LU Factorization: O(n^3)
#n pivots (p) (O(n))
#r elements = p+1, ..., n-1 that need reduction O(n)
#find multiplier = A[r][p] / A[p, p]
#subtract multiplier * subrow p' from subrow r' (O(n))
#efficient only when need to change b independent of A
L, U = cs111.LUfactorNoPiv(A)
print("L:", L, sep="\n")
print("U:", U, sep="\n")

y = cs111.Lsolve(L, b)
x = Usolve(U, y)
print("y:", y, sep="\n")
print("x:", x, sep="\n")

print("A @ x:", A @ x, sep="\n")

#residuals occur due to roundoff errors from finite bits
print("b - A @ x:", b - A @ x, sep="\n")
```

```
A:
[[0.4925 0.1339 0.2773 0.2897 0.4195 0.2725]
 [0.9347 0.4489 0.7792 0.7835 0.8027 0.0556]
 [0.4339 0.4488 0.6562 0.5433 0.7113 0.9749]
 [0.533  0.069  0.0811 0.8722 0.2141 0.348 ]
 [0.0383 0.4028 0.7716 0.9826 0.6065 0.0115]
 [0.0681 0.138  0.6969 0.8875 0.4497 0.3278]]
b:
[0.2649 0.3089 0.69   0.3181 0.3559 0.9925]
L:
[[  1.       0.       0.       0.       0.       0.    ]
 [  1.8981   1.       0.       0.       0.       0.    ]
 [  0.8811   1.6991   1.       0.       0.       0.    ]
 [  1.0823  -0.3901   6.8134   1.       0.       0.    ]
 [  0.0777   2.0157 -13.6064  -0.7124   1.       0.    ]
 [  0.1383   0.6139 -28.4805  -1.7213   1.6795   1.    ]]
U:
[[ 4.9245e-01  1.3394e-01  2.7732e-01  2.8973e-01  4.1946e-01  2.7252e-01]
 [ 0.0000e+00  1.9467e-01  2.5282e-01  2.3355e-01  6.4675e-03 -4.6167e-01]
 [ 0.0000e+00  0.0000e+00 -1.7675e-02 -1.0879e-01  3.3076e-01  1.5192e+00]
 [ 0.0000e+00  0.0000e+00  0.0000e+00  1.3910e+00 -2.4910e+00 -1.0478e+01]
 [ 0.0000e+00  0.0000e+00  0.0000e+00  0.0000e+00  3.2868e+00  1.4128e+01]
 [ 0.0000e+00  0.0000e+00  0.0000e+00  0.0000e+00  0.0000e+00  2.0786e+00]]
y:
[ 0.2649 -0.1939  0.7861 -5.3999  7.5747  1.4456]
x:
[ 0.164  -1.6724  1.6877  0.1306 -0.6848  0.6955]
A @ x:
[0.2649 0.3089 0.69   0.3181 0.3559 0.9925]
b - A @ x:
[-5.5511e-17  1.1102e-16  0.0000e+00 -1.3323e-15  3.2752e-15  4.6629e-15]
```

Partial Pivoting: use a permutation matrix to permute rows to solve a system with a 0 in a pivot. Complete pivoting occurs when both rows and columns are permuted.

Theorem: PA = LU for every non-singular square matrix A and, partial pivoting finds P, L, U.

New System of Interest: LUx = Pb

In [27]:
```
#NOTE: entire process is LU Solve, is O(n^3) dominated by factoring
A = np.array([[1, 1, 2], [1, 1, 3], [2, 3, 4]])
print("A:", A, sep="\n")

#LU factorization with partial pivoting
#searching for pivot is O(n^2)
#overall runtime dominanted by O(n^3)
#pivot finds the absmax of each column O(n) for each n columns
L, U, p = cs111.LUfactor(A)
print("L:", L, sep="\n")
print("U:", U, sep="\n")
print("L @ U:", L @ U, sep="\n")
print("A[p, :]:", A[p, :], sep="\n")

#non-singular, can solve for any solution
b = np.random.random(3)
print("b:", b, sep="\n")
print("b[p]:", b[p], sep="\n")

#solve for Ly = Pb => Ly = b[p]
y = cs111.Lsolve(L, b[p])
print("y:", y, sep="\n")

#solve for Ux = y
x = cs111.Usolve(U, y)
print("x:", x, sep="\n")

print("A @ x:", A @ x, sep="\n")
print("b - A @ x:", b - A @ x, sep="\n")
```

```
A:
[[1 1 2]
 [1 1 3]
 [2 3 4]]
L:
[[1.  0.  0. ]
 [0.5 1.  0. ]
 [0.5 1.  1. ]]
U:
[[ 2.   3.   4. ]
 [ 0.  -0.5  1. ]
 [ 0.   0.  -1. ]]
L @ U:
[[2. 3. 4.]
 [1. 1. 3.]
 [1. 1. 2.]]
A[p, :]:
[[2 3 4]
 [1 1 3]
 [1 1 2]]
b:
[0.1904 0.0018 0.1692]
b[p]:
[0.1692 0.0018 0.1904]
y:
[ 0.1692 -0.0828  0.1886]
x:
[ 0.7793 -0.2117 -0.1886]
A @ x:
[0.1904 0.0018 0.1692]
b - A @ x:
[0. 0. 0.]
```

In [28]:
```
#note doesnt work in this case because theres a 0 in pivot
L, U = cs111.LUfactorNoPiv(A)
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-28-50ee45210f6d> in <module>
      1 #note doesnt work in this case because theres a 0 in pivot
----> 2 L, U = cs111.LUfactorNoPiv(A)

~/cs111/LU.py in LUfactorNoPiv(A)
     27         # Update the rest of the matrix
     28         pivot = LU[piv_col, piv_col]
---> 29         assert pivot != 0., "pivot is zero, can't continue"
     30         for row in range(piv_col + 1, n):
     31             multiplier = LU[row, piv_col] / pivot

AssertionError: pivot is zero, can't continue
```

In [29]:
```
singular = np.array([[1,1,2], [1,1,3], [2,2,5]])

#rank of A (should be equal to len(A) where A is square to be non-singular)
print("Rank:", npla.matrix_rank(singular), sep="\n")

#singular matrices also have null vector v where Av = 0 such that v != 0
v = np.array([1, -1, 0])
print("v:", v, sep="\n")
print("singular @ v:", singular @ v, sep="\n")
```

```
Rank:
2
v:
[ 1 -1  0]
singular @ v:
[0 0 0]
```

In [30]:
```
cs111.LUsolve(singular, b)
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-30-886425fd7190> in <module>
----> 1 cs111.LUsolve(singular, b)

~/cs111/LU.py in LUsolve(A, b, pivoting)
    181
    182     # LU factorization
--> 183     L, U, p = LUfactor(A, pivoting=pivoting)
    184
    185     # Forward and back substitution

~/cs111/LU.py in LUfactor(A, pivoting)
    141         if pivoting:
    142             piv_row = piv_col + np.argmax(np.abs(LU[piv_col:, piv_col]))
--> 143             assert LU[piv_row, piv_col] != 0., "can't find nonzero pivot, m
atrix is singular"
    144             LU[[piv_col, piv_row], :]  = LU[[piv_row, piv_col], :]
    145             p[[piv_col, piv_row]]      = p[[piv_row, piv_col]]

AssertionError: can't find nonzero pivot, matrix is singular
```

Euclidean Norm: norm(v) = ||v|| = sqrt(v_1 + v_2 + ...)
Scaled Norm: norm(c * v) given constant c = |c| * ||v||
Error: x - x_exact (only useful if we know x_exact
Residual: b - Ax, determines how close x solves the problem
Residual Norm: ||b - Ax||, scalar quantity of residual error.
Relative Residual Norm: ||b - Ax|| / ||b||, normalized version of the residual normal
Note: when the residual or error is 0, then x is correct.

In [31]:
```
v = np.array([3, 1, 4, 1])
print("norm(v):", npla.norm(v))
print("norm(-2*v):", npla.norm(-2*v))

A = np.random.random((5, 5))
print("A:", A, sep="\n")

x_exact = np.ones(5)
b = A @ x_exact
print("b:", b, sep="\n")

x, r = cs111.LUsolve(A, b)

error = x - x_exact
print("error:", error, sep="\n")

print()

b = np.random.random(5)
print("b:", b, sep="\n")

x, rel_res = cs111.LUsolve(A, b)
print("x:", x, sep="\n")
print("residual:", b - A @ x, sep="\n")
print("residual norm:", npla.norm(b - A @ x), sep="\n")
print("relative residual norm:", npla.norm(b - A @ x) / npla.norm(b), sep="\n")
print("relative residual norm:", rel_res, sep="\n")
```

```
norm(v): 5.196152422706632
norm(-2*v): 10.392304845413264
A:
[[5.3543e-01 8.2182e-01 2.6132e-01 9.1640e-01 7.1384e-01]
 [4.5049e-01 9.4436e-02 4.9165e-01 4.7258e-01 2.5170e-01]
 [1.1378e-01 7.7464e-01 2.9115e-02 3.9401e-01 1.2459e-01]
 [7.7793e-01 3.1067e-01 1.7439e-01 2.6972e-01 3.9784e-01]
 [4.5888e-01 8.8686e-02 1.5431e-04 3.9458e-01 7.4056e-01]]
b:
[3.2488 1.7609 1.4361 1.9306 1.6829]
error:
[-1.5543e-15  1.7764e-15  3.3307e-15 -4.4409e-15  3.1086e-15]

b:
[0.7528 0.9251 0.314  0.8439 0.3135]
x:
[-0.4027  1.5305  3.8022 -3.0442  2.1107]
residual:
[-2.2204e-16  1.1102e-16  0.0000e+00  0.0000e+00  1.1102e-16]
residual norm:
2.7194799110210365e-16
relative residual norm:
1.7809573781994241e-16
relative residual norm:
1.7809573781994241e-16
```

In [32]:
```
#relative residual norm immune to scaling
km_per_lightyear = 9.461 * 10**12
AA = km_per_lightyear * A
bb = km_per_lightyear * b
xx, rr = cs111.LUsolve(AA, bb)

print("xx:", xx, sep="\n")
print("residual:", bb - AA @ xx, sep="\n")
print("residual norm:", npla.norm(bb - AA @ xx), sep="\n")
print("relative residual norm:", npla.norm(bb - AA @ xx) / npla.norm(bb), sep="\n")
print("relative residual norm:", rel_res, sep="\n")
```

```
xx:
[-0.4027  1.5305  3.8022 -3.0442  2.1107]
residual:
[ 0.001   0.002  -0.0024  0.      0.002 ]
residual norm:
0.003813598474563796
relative residual norm:
2.639766990195687e-16
relative residual norm:
1.7809573781994241e-16
```

Other Norms

Manhattan Distance: $||v||_1 = \sum_i |v_i|$
Max/Infinite Norm: $||v||_{\text{inf}} = max_i|v_i|$
p-norm: $||v||_p = (\sum_i |v_i|^p)^{1/p}$, approaches to the max norm as p approaches infinity
Zero Norm: $||v||_0$ = number of non-zeroes in v (not a real norm)

In [33]:
```
v = np.array([3, 1, 4, 0, 5])
print("2-norm:", npla.norm(v))
print("1-norm:", npla.norm(v, 1))
print("inf-norm:", npla.norm(v, np.inf))
print("10-norm:", npla.norm(v, 10))
print("0-norm:", npla.norm(v, 0))
```

```
2-norm: 7.14142842854285
1-norm: 13.0
inf-norm: 5.0
10-norm: 5.054008189891657
0-norm: 4.0
```

Jacobi Method

Note: Iterative Methods are preferable to direct solutions for large matrices.

Example:

$$\begin{pmatrix} 3 & 1 \\ -2 & 4 \end{pmatrix} * \begin{pmatrix} x_0 \\ x_1 \end{pmatrix} = \begin{pmatrix} 9 \\ 8 \end{pmatrix}$$

Idea: Solve for each x_i individually to make the ith equation satisfied.

Iteration Step:
$x_0^{k+1} = (9 - x_1^k)/3$
$x_1^{k+1} = (8 + 2 * x_0^k)/4$
with x^{k} being an initial guess

Matrix Form:

$$A : \begin{pmatrix} 3 & 1 \\ -2 & 4 \end{pmatrix} = C : \begin{pmatrix} 0 & 1 \\ -2 & 0 \end{pmatrix} + D : \begin{pmatrix} 3 & 0 \\ 0 & 4 \end{pmatrix}$$

$$x^{k+1} = (b - C * x^k) * D^{-1})$$

Conditions:

- the diagonal elements cannot be zero

In [34]:
```
A = np.array([[3, 1], [-2, 4]])
b = np.array([9, 8])
print(npla.solve(A, b)) #LU factorization with partial pivoting from numpy linear alg
print(cs111.LUsolve(A, b)) #also returns relative residual norm

# start with initial guess
x = np.zeros(2)
print("x:", x)
print("Relative Residual Norm:", npla.norm(b - A @ x) / npla.norm(b))

#try improve guess iteratively:
for i in range(10):
    x = np.array([(9 - x[1])/3, (8 + 2*x[0])/4])
    print("x:", x)
    print("Relative Residual Norm:", npla.norm(b - A @ x) / npla.norm(b))
```

```
[2. 3.]
(array([2., 3.]), 0.0)
x: [0. 0.]
Relative Residual Norm: 1.0
x: [3. 2.]
Relative Residual Norm: 0.5252257314388903
x: [2.3333 3.5   ]
Relative Residual Norm: 0.1666666666666666
x: [1.8333 3.1667]
Relative Residual Norm: 0.08753762190648182
x: [1.9444 2.9167]
Relative Residual Norm: 0.027777777777777832
x: [2.0278 2.9722]
Relative Residual Norm: 0.014589603651080349
x: [2.0093 3.0139]
Relative Residual Norm: 0.00462962962962965
x: [1.9954 3.0046]
Relative Residual Norm: 0.00243160060851345
x: [1.9985 2.9977]
Relative Residual Norm: 0.0007716049382715942
x: [2.0008 2.9992]
Relative Residual Norm: 0.0004052667680856294
x: [2.0003 3.0004]
Relative Residual Norm: 0.0001286008230453045
```

In [35]:
```
# Matrix View of Jacobi
d = A.diagonal()
D = np.diag(d)
C = A - D

print("A:", A, sep="\n")
print("C:", C, sep="\n")
print("D:", D, sep="\n")
print("d:", d)

# start with initial guess
x = np.zeros(2)
print("x:", x)
print("Relative Residual Norm:", npla.norm(b - A @ x) / npla.norm(b))

#try improve guess iteratively using matrix:
#stationary iterative method: C and d do not change with each iteration
for i in range(10):
    x = (b - C @ x) / d
    print("x:", x)
    print("Relative Residual Norm:", npla.norm(b - A @ x) / npla.norm(b))
```

```
A:
[[ 3  1]
 [-2  4]]
C:
[[ 0  1]
 [-2  0]]
D:
[[3 0]
 [0 4]]
d: [3 4]
x: [0. 0.]
Relative Residual Norm: 1.0
x: [3. 2.]
Relative Residual Norm: 0.5252257314388903
x: [2.3333 3.5   ]
Relative Residual Norm: 0.1666666666666666
x: [1.8333 3.1667]
Relative Residual Norm: 0.08753762190648182
x: [1.9444 2.9167]
Relative Residual Norm: 0.027777777777777832
x: [2.0278 2.9722]
Relative Residual Norm: 0.014589603651080349
x: [2.0093 3.0139]
Relative Residual Norm: 0.00462962962962965
x: [1.9954 3.0046]
Relative Residual Norm: 0.00243160060851345
x: [1.9985 2.9977]
Relative Residual Norm: 0.0007716049382715942
x: [2.0008 2.9992]
Relative Residual Norm: 0.0004052667680856294
x: [2.0003 3.0004]
Relative Residual Norm: 0.0001286008230453045
```

In [36]:
```
# FROM cs111 library
def Jsolve(A, b, tol = 1e-8, max_iters = 1000, callback = None):
    """Solve a linear system Ax = b for x by the Jacobi iterative method.
    Parameters:
      A: the matrix.
      b: the right-hand side vector.
      tol: the relative residual at which to stop iterating.
      max_iters: the maximum number of iterations to do.
      callback: a user function to call at every iteration.
        The callback function has arguments 'x', 'iteration', and 'residual'
    Outputs (in order):
      x: the computed solution
      rel_res: list of relative residual norms at each iteration.
        The number of iterations actually done is len(rel_res) - 1
    """
    # Check the input
    m, n = A.shape
    assert m == n, "matrix must be square"
    bn, = b.shape
    assert bn == n, "rhs vector must be same size as matrix"

    # Split A into diagonal D plus off-diagonal C
    d = A.diagonal()          # diagonal elements of A as a vector
    # D = np.diag(d)          # diagonal of A as a matrix -- DON'T DO THIS, IT CREATES
    C = A.copy()
    C.setdiag(np.zeros(n))    # A without the diagonal

    # Initial guess: x = 0
    x = np.zeros(n)

    # Vector of relative residuals
    # Relative residual is norm(residual)/norm(b)
    # Intitial residual is b - Ax for x=0, or b
    rel_res = [1.0]

    # Call user function if specified
    if callback is not None:
        callback(x = x, iteration = 0, residual = 1)

    # Iterate
    for k in range(1, max_iters+1):
        # New x
        x = (b - C @ x) / d

        # Record relative residual
        this_rel_res = npla.norm(b - A @ x) / npla.norm(b)
        rel_res.append(this_rel_res)

        # Call user function if specified
        if callback is not None:
            callback(x = x, iteration = k, residual = this_rel_res)

        # Stop if within tolerance
        if this_rel_res <= tol:
            break

    return (x, rel_res)
```

Jacobi Issues

In [37]:
```
A = np.array([[1, 2], [3, 4]])
b = A @ np.array([1, 1])
print("A:", A, sep="\n")
print("b:", b, sep="\n")
print(cs111.LUsolve(A, b)) #also returns relative residual norm

# start with initial guess
x = np.zeros(2)
print("x:", x)
print("Relative Residual Norm:", npla.norm(b - A @ x) / npla.norm(b))

# Matrix View of Jacobi
d = A.diagonal()
D = np.diag(d)
C = A - D

print("A:", A, sep="\n")
print("C:", C, sep="\n")
print("D:", D, sep="\n")
print("d:", d)

#try improve guess iteratively using matrix:
#stationary iterative method: C and d do not change with each iteration
for i in range(10):
    x = (b - C @ x) / d
    print("x:", x)
    print("Relative Residual Norm:", npla.norm(b - A @ x) / npla.norm(b))

#ISSUE: Matrix results in exploding residuals
```

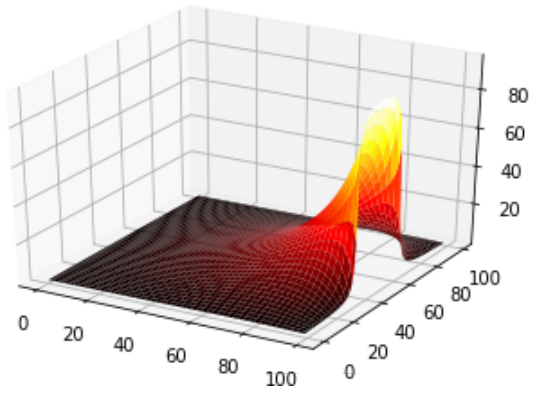
```
A:
[[1 2]
 [3 4]]
b:
[3 7]
(array([1., 1.]), 0.0)
x: [0. 0.]
Relative Residual Norm: 1.0
A:
[[1 2]
 [3 4]]
C:
[[0 2]
 [3 0]]
D:
[[1 0]
 [0 4]]
d: [1 4]
x: [3.   1.75]
Relative Residual Norm: 1.2679742192527634
x: [-0.5 -0.5]
Relative Residual Norm: 1.5
x: [4.    2.125]
Relative Residual Norm: 1.9019613288791453
x: [-1.25 -1.25]
Relative Residual Norm: 2.25
x: [5.5    2.6875]
Relative Residual Norm: 2.852941993318718
x: [-2.375 -2.375]
Relative Residual Norm: 3.3749999999999996
x: [7.75   3.5312]
Relative Residual Norm: 4.279412989978076
```

```
x: [-4.0625 -4.0625]
Relative Residual Norm: 5.0625
x: [11.125   4.7969]
Relative Residual Norm: 6.419119484967116
x: [-6.5938 -6.5938]
Relative Residual Norm: 7.59375
```

Jacobi on the Temperature Problem

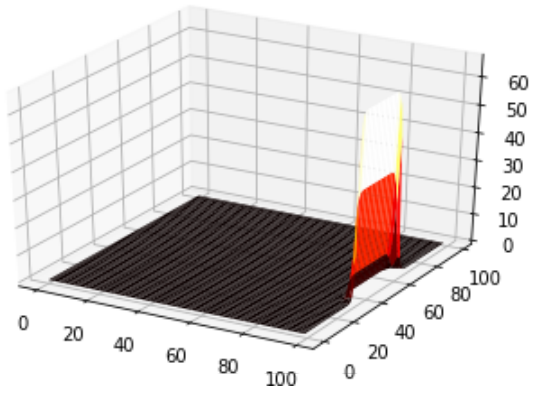
In [38]:
```
# Optimal Solution
k = 100
A = cs111.make_A(k)
b = cs111.make_b(k, right=cs111.radiator(k))
t = scipy.sparse.linalg.spsolve(A, b)
T = t.reshape(k, k)
X, Y = np.meshgrid(range(k), range(k))
fig = plt.figure()
ax = fig.gca(projection='3d')
ax = fig.gca()
ax.plot_surface(X, Y, T, cmap=cm.hot)
```

Out[38]:
```
<mpl_toolkits.mplot3d.art3d.Poly3DCollection at 0x7f9c0d461c10>
```



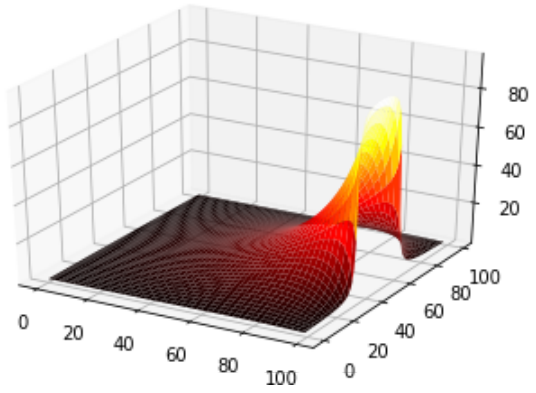
In [39]:
```
# Jacobi with 10 Iterations
t, resvec = cs111.Jsolve(A, b, max_iters = 10)
T = t.reshape(k, k)
X, Y = np.meshgrid(range(k), range(k))
fig = plt.figure()
ax = fig.gca(projection='3d')
ax = fig.gca()
ax.plot_surface(X, Y, T, cmap=cm.hot)
```

Out[39]:
```
<mpl_toolkits.mplot3d.art3d.Poly3DCollection at 0x7f9c0d479510>
```



In [40]:
```
# Jacobi with 10000 Iterations
t, resvec = cs111.Jsolve(A, b, max_iters = 10000)
T = t.reshape(k, k)
X, Y = np.meshgrid(range(k), range(k))
fig = plt.figure()
ax = fig.gca(projection='3d')
ax = fig.gca()
ax.plot_surface(X, Y, T, cmap=cm.hot)
```

Out[40]:
```
<mpl_toolkits.mplot3d.art3d.Poly3DCollection at 0x7f9c0db13a90>
```



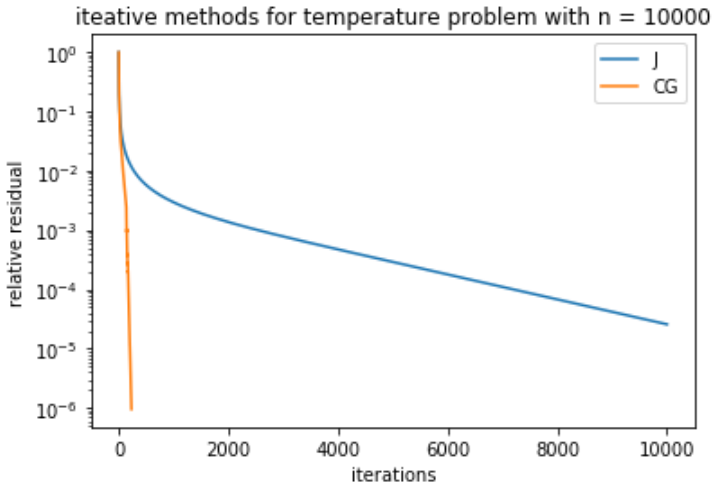
Conjugate Gradient Iteration (SPD Matrices Only)

In [41]:
```
# Congugant gradient: more powerful iterative method
# Matrix A must be Symmetric Positive Definite Matrix
# Moves in search direction d equal to direction of residuals b - Ax
t, resvec = cs111.CGsolve(A, b, max_iters = 200)
```

In [42]:
```
%matplotlib inline
plt.figure()

(xJ, resvecJ) = cs111.Jsolve(A, b, tol=1e-6, max_iters = 10000)
print("Jacobi iters:", len(resvecJ) - 1)
print("Last Relative Residual Norm:", resvecJ[-1])
print("Computed Relative Resiual Norm:", npla.norm(A @ xJ - b) / npla.norm(b))
plt.semilogy(resvecJ, label="J")

(xCG, resvecCG) = cs111.CGsolve(A, b, tol=1e-6, max_iters = 10000)
print("Jacobi iters:", len(resvecCG) - 1)
print("Last Relative Residual Norm:", resvecCG[-1])
print("Computed Relative Resiual Norm:", npla.norm(A @ xCG - b) / npla.norm(b))
plt.semilogy(resvecCG, label="CG")

plt.legend()
plt.xlabel("iterations")
plt.ylabel("relative residual")
plt.title("iteative methods for temperature problem with n = {}".format(A.shape[0]))
```

```
Jacobi iters: 10000
Last Relative Residual Norm: 2.575758711403401e-05
Computed Relative Resiual Norm: 2.575758711403401e-05
Jacobi iters: 232
Last Relative Residual Norm: 9.547478926650693e-07
Computed Relative Resiual Norm: 9.547478926650693e-07
```

Out[42]:
```
Text(0.5, 1.0, 'iteative methods for temperature problem with n = 10000')
```



In [43]:
```
# FROM CS111 Library
def CGsolve(A, b, tol = 1e-8, max_iters = 1000, callback = None):
    """Solve a linear system Ax = b for x by the conjugate gradient iterative method.
    Parameters:
      A: the matrix.
      b: the right-hand side vector.
      tol: the relative residual at which to stop iterating.
      max_iters: the maximum number of iterations to do.
      callback: a user function to call at every iteration.
        The callback function has arguments 'x', 'iteration', and 'residual'
    Outputs (in order):
      x: the computed solution
      rel_res: list of relative residual norms at each iteration.
        The number of iterations actually done is len(rel_res) - 1
    """
    # Check the input
    m, n = A.shape
    assert m == n, "matrix must be square"
    bn, = b.shape
    assert bn == n, "rhs vector must be same size as matrix"

    # Make the matrix use the sparse csr data structure
    A = scipy.sparse.csr_matrix(A)

    # Initial guess: x = 0
    x = np.zeros(n)

    # Initial residual: r = b - A@0 = b
    r = b

    # Initial step is in direction of residual.
    d = r

    # Squared norm of residual
    rtr = r.T @ r

    # Vector of relative residuals
    # Relative residual is norm(residual)/norm(b)
    # Intitial residual is b - Ax for x=0, or b
    rel_res = [1.0]

    # Call user function if specified
    if callback is not None:
        callback(x = x, iteration = 0, residual = 1)

    # Iterate
    # One Matrix-Vector Multiplication => O(n^2)
    # Note: if A is sparse, time = O(#nonzeroes in A)
    # Two Dot Products => O(n)
    # Matrix Addition => O(n)
    # Four Vectors of Working Storage
    for k in range(1, max_iters+1):
        Ad = A @ d
        alpha = rtr / (d.T @ Ad)  # Length of step
        x = x + alpha * d         # Update x to new x
        r = r - alpha * Ad        # Update r to new residual
        rtrold = rtr
        rtr = r.T @ r
        beta = rtr / rtrold
        d = r + beta * d          # Update d to new step direction

        # Record relative residual
        this_rel_res = npla.norm(b - A @ x) / npla.norm(b)
```

```
        rel_res.append(this_rel_res)

        # Call user function if specified
        if callback is not None:
            callback(x = x, iteration = k, residual = this_rel_res)

        # Stop if within tolerance
        if this_rel_res <= tol:
            break

    return (x, rel_res)
```

In [44]:
```
#Equivalent Form:
scipy.sparse.linalg.cg?
```

Matrix Size

Rank: number of linearly independent column vectors of a matrix
Theroem: rank of a matrix is equal to the number of linearly independent rows, even if the matrix is non square
Null Vector: A @ v results in the null vector
Norm: $M = ||A|| = max(||Av||/||v||)$ for all non-zero vector v, that is the maximum stretch
Note: any vector norm induces a matrix norm (i.e., 2-vector norm induces a 2-vector norm
Condition Number: $\kappa(A) = M/m$ where M is the max norm and m is the min norm (stretch)
Ill-Conditioned: $\kappa(A)$ is large and tends to infinity (i.e., m = 0, meaning there exists 1 null-vector $||Av|| = 0$)
Well-Conditioned: $\kappa(A)$ is small and tends to 1
Note: If A is square and nonsingular, $\kappa(A) = ||A||||A^{-1}||$ where A^{-1} unstretches v

In [45]:
```
# RANK

#spans 2D subspace in 3D
A = np.array([[1, 4, 7], [2, 5, 8], [3, 6, 9]])
print("Middle column is avg of other two columns:", (A[:,1] == (A[:,0] + A[:,2]) / 2)

#contains a null vector
v = np.array([-1/2, 1, -1/2])
print("A @ v:", A @ v)

#A singular matrix has rank < dimension and a null vector
#Note: nonsingular matrices have rank = dimension
print("Rank of A:", npla.matrix_rank(A))

#Theorem:
print("Rank of A.T:", npla.matrix_rank(A.T))

#spans 1D subspace in 3D
A = np.array( [[1, 2, 3], [2, 4, 6], [3, 6, 9], [4, 8, 12]])
print("Rank of A:", npla.matrix_rank(A))
print("Rank of A.T:", npla.matrix_rank(A.T))

#Matrix cannot have dimension < rank:
A = np.random.random( (4, 7) )
print("Rank of A:", npla.matrix_rank(A))
print("Rank of A.T:", npla.matrix_rank(A.T))
```

```
Middle column is avg of other two columns: True
A @ v: [0. 0. 0.]
Rank of A: 2
Rank of A.T: 2
Rank of A: 1
Rank of A.T: 1
Rank of A: 4
Rank of A.T: 4
```

In [46]:
```
# NORM
A = np.array( [[2,-1,1],[1,0,1],[3,-1,4]] )

print("2-matrix norm:", npla.norm(A, 2)) #O(n^3) using SVD

#equals largest column abs sum
print("1-matrix norm:", npla.norm(A, 1))

#equals largest row abs sum
print("inf-matrix norm:", npla.norm(A, np.inf))
```

```
2-matrix norm: 5.722926953325028
1-matrix norm: 6.0
inf-matrix norm: 8.0
```

In [47]:
```
# CONDITION NUMBER

# identity matrix
A = np.eye( 5 )
print("Condition # of I:", npla.cond( A, 2 ))

# singular matrix
A = np.array( [[1, 4, 7], [2, 5, 8], [3, 6, 9]] )
print("Condition # of Singular:", npla.cond( A, 2 ))

# diagonal matrix
A = np.diag( [1,2,3,4,5,6,7,8] )
print("Condition # of Diagonal:", npla.cond( A, 2 ))

# random matrix
A = np.random.random( (5,5) )
print("Condition # of Rand (2 norm):", npla.cond( A, 2 ))
print("Condition # of Rand (1 norm):", npla.cond( A, 1 ))
print("Condition # of Rand (inf norm):", npla.cond( A, np.inf ))

# scaling does not change condition number
#(scaling cancelled out in max divided by min)
print("Norm: ", npla.norm(A, 2))
print("Scaled Norm: ", npla.norm(10 * A, 2))
print("Condition #:", npla.cond( A, 2 ))
print("Scaled Condition #:", npla.cond(10 * A, 2 ))
```

```
Condition # of I: 1.0
Condition # of Singular: 3.3434160864560614e+17
Condition # of Diagonal: 8.0
Condition # of Rand (2 norm): 42.83387247447056
Condition # of Rand (1 norm): 100.43593297578525
Condition # of Rand (inf norm): 50.89615711251314
Norm:  2.5390324520687972
Scaled Norm:  25.390324520687972
Condition #: 42.83387247447056
Scaled Condition #: 42.83387247447057
```

Symmetric Positive Definite Matrices

Symmetry: a matrix A is symmetric when A = A.T
Positive Definite: all eigenvalues and pivots are positive; additioanlly, x.T @ A @ x > 0 for every nonzero vector x
Theorem: If A is any matrix, square or not, with full column rank (i.e. all the columns are linearly independent), then $A^T A$ is SPD.
Note: Every entry A.T @ A is the dot product of two columns in A.

In [48]:
```
A = cs111.make_A( 2 ).toarray()

# Symmetric:
print("Symmetry:", np.all( A == A.T ))

# Positive definite: LU without partial pivoting works, and all pivots are␣→positive.
L, U = cs111.LUfactorNoPiv( A )
print("U Diagonal:", U.diagonal())

# Positive definite: Eigenvalues are all positive.
vals, vecs = npla.eig( A )
print("Eigenvalues:", vals)

# Positive definite: x.T @ A @ x > 0 for every nonzero vector x.
x = np.random.randn( 4 )

# Full column rank results in A^T * A being SPD
A = np.random.random( (8, 4) )
print("Rank:", npla.matrix_rank( A ))

B = A.T @ A
print("Symmetry:", np.all(B == B.T ))

vals, vecs = npla.eig( B )
print("Eigenvalues:", vals)

L, U = cs111.LUfactorNoPiv( B )
print("U Diagonal:", U.diagonal())

# dot product to prove symmetric matrix:
print("Col 2 DOT Col 0:", A[:,2].dot(A[:,0]))
print("Element in B[2, 0]:", B[2, 0])
print("Element in B[0, 2]:", B[0, 2])
```

```
Symmetry: True
U Diagonal: [4.     3.75   3.7333 3.4286]
Eigenvalues: [2. 4. 6. 4.]
Rank: 4
Symmetry: True
Eigenvalues: [9.0133 1.21   0.5009 0.1637]
U Diagonal: [2.3265 1.3543 0.3972 0.7148]
Col 2 DOT Col 0: 1.772382779970024
Element in B[2, 0]: 1.772382779970024
Element in B[0, 2]: 1.772382779970024
```

Error and Residual

Theorem: Conjugate Gradient converges to a fixed tolerance on SPD A in at most $O(\sqrt{\kappa_2(A)})$
Run Time of Conjugate Gradient: When a matrix is sparse, the run time is $O(nnz(A) * \sqrt{\kappa_2(A)})$ where nnz is number of nonzeroes; When a matrix is dense, the run time is $O(n^2 * \sqrt{\kappa_2(A)})$
Note: the error is zero if and only if the residual is 0
Theorem: $||e||/||x_{exact}|| \leq \kappa(A) * ||r||/||b||$
where $||e||/||x_{exact}||$ is the relative error norm and $||r||/||b||$ is the relative residual norm
Note: Relative Residual Norm does not provide insight on goodness of solution since we're bounded by condition number

In [49]:
```
## GOOD EXAMPLE
A = np.random.random( (10,10) )
x_exact = np.ones( A.shape[1] )
b = A @ x_exact

x = npla.solve( A, b )
print( "x:", x )

residual = b - A @ x
error = x_exact - x
print( "n:", A.shape[1] )
print( "relative residual norm:", npla.norm( residual ) / npla.norm( b ) )
print( "relative error norm:", npla.norm( error ) / npla.norm( x_exact ) )
print( "condition number of A:", npla.cond(A, 2) )

#relative error norm is bounded:
print( "Bounded?:", npla.norm( error ) / npla.norm( x_exact ) <= npla.norm( residual
```

```
x: [1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
n: 10
relative residual norm: 9.325388119872257e-17
relative error norm: 3.1110026086344394e-15
condition number of A: 108.56405385471643
Bounded?: True
```

In [50]:
```
## ILL CONDITIONED EXAMPLE
def hilbert( n ):
    """
    n-by-n Hilbert matrix, a famous example of an ill-conditioned matrix
    """
    A = np.zeros( (n,n) )
    for i in range( n ):
        for j in range( n ):
            A[i,j] = 1 / (i + j + 1)

    return A
```

In [51]:
```
A = hilbert( 10 )
x_exact = np.ones( A.shape[1] )
b = A @ x_exact
print("A:", A, sep="\n") #A is SPD
print("b:", b, sep="\n")

x = npla.solve( A, b )
print( "x:", x )

residual = b - A @ x
error = x_exact - x
print( "n:", A.shape[1] )
print( "relative residual norm:", npla.norm( residual ) / npla.norm( b ) )
print( "relative error norm:", npla.norm( error ) / npla.norm( x_exact ) )
print( "condition number of A:", npla.cond(A, 2) )

#relative error norm is bounded:
print( "Bounded?:", npla.norm( error ) / npla.norm( x_exact ) <= npla.norm( residual
```

```
A:
[[1.     0.5    0.3333 0.25   0.2    0.1667 0.1429 0.125  0.1111 0.1   ]
 [0.5    0.3333 0.25   0.2    0.1667 0.1429 0.125  0.1111 0.1    0.0909]
 [0.3333 0.25   0.2    0.1667 0.1429 0.125  0.1111 0.1    0.0909 0.0833]
 [0.25   0.2    0.1667 0.1429 0.125  0.1111 0.1    0.0909 0.0833 0.0769]
 [0.2    0.1667 0.1429 0.125  0.1111 0.1    0.0909 0.0833 0.0769 0.0714]
 [0.1667 0.1429 0.125  0.1111 0.1    0.0909 0.0833 0.0769 0.0714 0.0667]
 [0.1429 0.125  0.1111 0.1    0.0909 0.0833 0.0769 0.0714 0.0667 0.0625]
 [0.125  0.1111 0.1    0.0909 0.0833 0.0769 0.0714 0.0667 0.0625 0.0588]
 [0.1111 0.1    0.0909 0.0833 0.0769 0.0714 0.0667 0.0625 0.0588 0.0556]
 [0.1    0.0909 0.0833 0.0769 0.0714 0.0667 0.0625 0.0588 0.0556 0.0526]]
b:
[2.929  2.0199 1.6032 1.3468 1.1682 1.0349 0.9307 0.8467 0.7773 0.7188]
x: [1.     1.     1.     1.     1.     0.9999 1.0002 0.9998 1.0001 1.    ]
n: 10
relative residual norm: 1.2019622663668722e-16
relative error norm: 8.67039023709691e-05
condition number of A: 16024416992541.715
Bounded?: True
```

In [52]:
```
A = hilbert( 20 )
x_exact = np.ones( A.shape[1] )
b = A @ x_exact

x = npla.solve( A, b )
print( "x:", x )

residual = b - A @ x
error = x_exact - x
print( "n:", A.shape[1] )
print( "relative residual norm:", npla.norm( residual ) / npla.norm( b ) )
print( "relative error norm:", npla.norm( error ) / npla.norm( x_exact ) )
print( "condition number of A:", npla.cond(A, 2) )

#relative error norm is bounded:
print( "Bounded?:", npla.norm( error ) / npla.norm( x_exact ) <= npla.norm( residual
```

```
x: [  1.       1.       1.0003   1.0017   0.903    2.0862  -5.2096  21.837
 -40.6083  45.1452  -9.372  -19.968   -2.7841  49.1131 -33.8648 -12.0984
  24.971   -3.2926  -3.5734   2.7136]
n: 20
relative residual norm: 4.1614166585757906e-16
relative error norm: 21.253725525646193
condition number of A: 1.3553657908688225e+18
Bounded?: True
```

Orthogonality

Dot Product: $x \cdot y = x^T y = \sum_{i=0}^{n-1} x_i y_i$
Theorem: $x^T x = ||x||_2^2$
Theorem: If $x^T y = 0$ and x, y != 0, then x and y are orthogonal vectors
Orthogonal Matrix: Matrix A is orthogonal if it is square and $A^T A = AA^T = I$
Conclusions about Orthogonal Matrix:

- Columns are perpendicular and unit length (since when i != j, the result in the identity matrix = 0)
- Rows are perpendicular
- A^T is also orthogonal
- $A^{-1} = A^T$ when A is orthogonal
- Identity and permutation matrices are orthogonal

Theorem: Orthogonal matrices do not change the length (nor angle) of a vector: $\forall_{v,Q} ||Qv||_2 = ||v||_2$
Conclusions:

- Orthogonal matrices have 2-norm equal to 1: $||Q||_2 = max_{v!=0} ||Qv||_2 / ||v||_2 = 1$
- Condition number: $\kappa_2(Q) = ||Q||_2 * ||Q^{-1}||_2 = ||Q||_2 * ||Q^T||_2 = 1$
- Solving $Qx = b$: $x = Q^T b$ => $||x||_2 = ||b||_2$, with two norm error equal to residual

Frobenius Norm: $||A||_F = \sqrt{\sum_{i,j} a_{i,j}^2}$ (Not as useful for numerical analyses as 2-norm, but much easier to compute

In [53]:
```
x = np.array([1, 1, 2, 1])
y = np.array([1, 2, -1, -1])
print("Dot Product:", np.dot(x, y))
print("Dot Product:", x.T @ y)
print("Dot with Self:", np.dot(x, x))
print("Two Norm Squared:", npla.norm(x, 2) ** 2)
```

```
Dot Product: 0
Dot Product: 0
Dot with Self: 7
Two Norm Squared: 7.000000000000001
```

In [54]:
```
Q = cs111.random_orthog(5)
print ("Q.T @ Q:", Q.T @ Q, sep="\n")
print("norm(Q):", npla.norm(Q, 2))
print("cond(Q):", npla.cond(Q, 2))

I = np.eye(5)
print("Error Norm:", npla.norm(Q.T @ Q - I, 2))

v = np.random.random(5)
print("v:", v)
print("norm(v):", npla.norm(v, 2))

w = Q @ v
print("w:", w)
print("norm(w):", npla.norm(w, 2))

x_exact = np.ones(5)
b = Q @ x_exact
x = Q.T @ b
error = x_exact - x
residual = b - Q @ x
print("Relative Residual Norm:", npla.norm(residual, 2) / npla.norm(b, 2))
print("Relative Error Norm:", npla.norm(error, 2) / npla.norm(x_exact, 2))
```

```
Q.T @ Q:
[[ 1.0000e+00 -2.4878e-17  2.4955e-18  1.5734e-16 -1.9491e-16]
 [-2.4878e-17  1.0000e+00  1.5336e-16  1.8875e-16  7.3842e-17]
 [ 2.4955e-18  1.5336e-16  1.0000e+00  1.3732e-16  2.3796e-17]
 [ 1.5734e-16  1.8875e-16  1.3732e-16  1.0000e+00  9.1444e-17]
 [-1.9491e-16  7.3842e-17  2.3796e-17  9.1444e-17  1.0000e+00]]
norm(Q): 1.0000000000000002
cond(Q): 1.0000000000000007
Error Norm: 5.063984862890652e-16
v: [0.6769 0.3058 0.429  0.7743 0.3465]
norm(v): 1.2063808128107758
w: [ 0.0064 -1.0361  0.2541 -0.1297 -0.5481]
norm(w): 1.206380812810776
Relative Residual Norm: 4.845713351197033e-16
Relative Error Norm: 4.328446199157272e-16
```

QR Factorization

Theorem: Every matrix A in any dimension can be written as $A = QR$ with Q being orthogonal and R being upper triangular.

In [55]:
```
# scipy.linalg.qr?

# SOLVING LINEAR SYSTEM WITH QR FACTORIZATION: useful for nonsquare system
A = np.random.random((7, 4))
print("A:", A)

Q, R = scipy.linalg.qr(A) #full size factorization
print("Q:", Q)
print("R:", R)

# Frobenius Norms
I = np.eye(7)
print("Frobenius Norm:", npla.norm(Q.T @ Q - I)) #close to zero
print("Frobenius Norm:", npla.norm(Q @ R - A)) #close to zero

Q, R = scipy.linalg.qr(A, mode="economic") #economic size factorization
print("Q:", Q) # not square, not orthogonal => columns are orthogonal
print("R:", R) # true upper triangular square matrix

#close to zero, same error as full size, but cheaper since stops once R is square
print("Frobenius Norm:", npla.norm(Q @ R - A))

x_exact = np.ones(4)
b = A @ x_exact
print("b:", b)

# Ax = b
# A = QR
# R x = Q^T b
x = cs111.Usolve(R, Q.T @ b)
print("x:", x)

error = x_exact - x
residual = b - A @ x

# close to zero
print("Relative Residual Norm:", npla.norm(residual, 2) / npla.norm(b, 2))
print("Relative Error Norm:", npla.norm(error, 2) / npla.norm(x_exact, 2))
```

```
A: [[6.9722e-01 2.7664e-01 3.7982e-01 5.7193e-02]
 [8.9395e-01 6.1554e-01 6.3687e-04 5.2717e-01]
 [1.6553e-01 4.7437e-01 3.4809e-01 7.5347e-01]
 [3.7230e-01 7.4282e-01 4.5735e-01 2.1001e-02]
 [6.5772e-01 4.1609e-01 9.0709e-01 1.4398e-01]
 [8.1685e-01 8.2412e-02 1.3600e-01 1.5596e-01]
 [5.5062e-01 4.7305e-01 9.2399e-01 7.5154e-01]]
Q: [[-0.4127  0.1754 -0.0591 -0.2236 -0.5483 -0.6051 -0.2805]
 [-0.5291 -0.1054  0.6348  0.1968  0.4127 -0.2587  0.1732]
 [-0.098  -0.4758  0.0031  0.5645 -0.1593  0.1559 -0.629 ]
 [-0.2204 -0.6598  0.1234 -0.5599 -0.2546  0.3051  0.1717]
 [-0.3893 -0.0309 -0.5339 -0.2932  0.6059  0.0211 -0.33  ]
 [-0.4835  0.5115  0.0888  0.0589 -0.2213  0.6655 -0.0373]
 [-0.3259 -0.1838 -0.5341  0.4355 -0.151  -0.0795  0.5966]]
R: [[-1.6895 -1.006  -1.012  -0.7574]
 [ 0.     -0.7898 -0.5292 -0.4807]
 [ 0.      0.     -0.9303 -0.1283]
 [ 0.      0.      0.      0.7988]
 [ 0.      0.      0.      0.    ]
 [ 0.      0.      0.      0.    ]
 [ 0.      0.      0.      0.    ]]
Frobenius Norm: 1.3065691544810372e-15
Frobenius Norm: 1.4880957734169956e-15
```

```
Q: [[-0.4127  0.1754 -0.0591 -0.2236]
 [-0.5291 -0.1054  0.6348  0.1968]
 [-0.098  -0.4758  0.0031  0.5645]
 [-0.2204 -0.6598  0.1234 -0.5599]
 [-0.3893 -0.0309 -0.5339 -0.2932]
 [-0.4835  0.5115  0.0888  0.0589]
 [-0.3259 -0.1838 -0.5341  0.4355]]
R: [[-1.6895 -1.006  -1.012  -0.7574]
 [ 0.     -0.7898 -0.5292 -0.4807]
 [ 0.      0.     -0.9303 -0.1283]
 [ 0.      0.      0.      0.7988]]
Frobenius Norm: 1.4880957734169956e-15
b: [1.4109 2.0373 1.7415 1.5935 2.1249 1.1912 2.6992]
x: [1. 1. 1. 1.]
Relative Residual Norm: 2.847389300790462e-16
Relative Error Norm: 8.41868291085074e-16
```

Least Squares

Surveyor Problem: first surveyor measures heights of mountains and second surveyor measures differences of heights of mountains. Since there are more equations than unknowns, the system becomes inconsistent with no solutions; must use least squares to satisfy as many constraints as possible of this over-determined system.
Least-Squares Technique: given a m x n matrix A, n x 1 vector x and m x 1 vector b, the least square solution minimizes $||r||_2 = ||b - Ax||_2$.
Transforming this equation, we get $||Q^T r||_2 = ||Q^T b - Q^T Ax||_2 = ||Q^T b - Rx||_2$ because orthogonal matrices preserve norms.
$x = min||Q^T b - Rx||_2$
Note: Use Usolve (since R is upper triangular) with the economy-size matrix R to get a perfect solution for the nonzero square (n x n) portion of the system.
Fitting a Line: $min(\sum y(t_i) - c_i)$ where c_i is observed and $y(t_i)$ is predicted. Since the line is modelled by $y = x_0 + x_1 * t$, the corresponding matrix A is n x 2, with constant 1 in the x_0 column and t_i in the x_1 column, with the solution $b_i = c_i$.

In [56]:
```
#first 3 rows = first surveyor
#last 3 rows = second surveyor
A = np.array( [[1,0,0], [0,1,0], [0,0,1], [-1,1,0], [-1,0,1], [0,-1,1]] )
b = np.array( [1237, 1941, 2417, 711, 1177, 475] )

print( "A:", A, sep="\n" )
print( "b:", b )

# economic QR factorization
Q, R = spla.qr( A, mode="economic" )
print( "Q shape:", Q.shape )
print( "Q:", Q, sep="\n" )
print( "R shape:", R.shape )
print( "R:", R, sep="\n" )

print("Frobenius Norm:", npla.norm( Q @ R - A ))

# RHS
print("b shape:", b.shape)
print("Q.T @ b", Q.T @ b)

# Perfect Solution for Square Portion
x = cs111.Usolve( R, Q.T @ b )
print("x:", x)

# Residuals
print("Residual:", b - A @ x)
print( "Relative Residual Norm:", npla.norm( b - A @ x ) / npla.norm( b ) )

# Check for First Surveyor
first_surveyor_x = np.array( [1237, 1941, 2417] )
relres = npla.norm( b - A @ first_surveyor_x ) / npla.norm( b )
print( "First surveyor's relres: ", relres )
```

```
A:
[[ 1  0  0]
 [ 0  1  0]
 [ 0  0  1]
 [-1  1  0]
 [-1  0  1]
 [ 0 -1  1]]
b: [1237 1941 2417  711 1177  475]
Q shape: (6, 3)
Q:
[[-5.7735e-01 -2.0412e-01 -3.5355e-01]
 [-0.0000e+00 -6.1237e-01 -3.5355e-01]
 [-0.0000e+00 -0.0000e+00 -7.0711e-01]
 [ 5.7735e-01 -4.0825e-01  2.7756e-17]
 [ 5.7735e-01  2.0412e-01 -3.5355e-01]
 [-0.0000e+00  6.1237e-01 -3.5355e-01]]
R shape: (3, 3)
R:
[[-1.7321  0.5774  0.5774]
 [ 0.     -1.633   0.8165]
 [ 0.      0.     -1.4142]]
Frobenius Norm: 4.832017418470168e-16
b shape: (6,)
Q.T @ b [  375.855 -1200.25  -3416.74 ]
x: [1236. 1943. 2416.]
Residual: [ 1. -2.  1.  4. -3.  2.]
Relative Residual Norm: 0.001624903391484253
First surveyor's relres:  0.002109694296525749
```

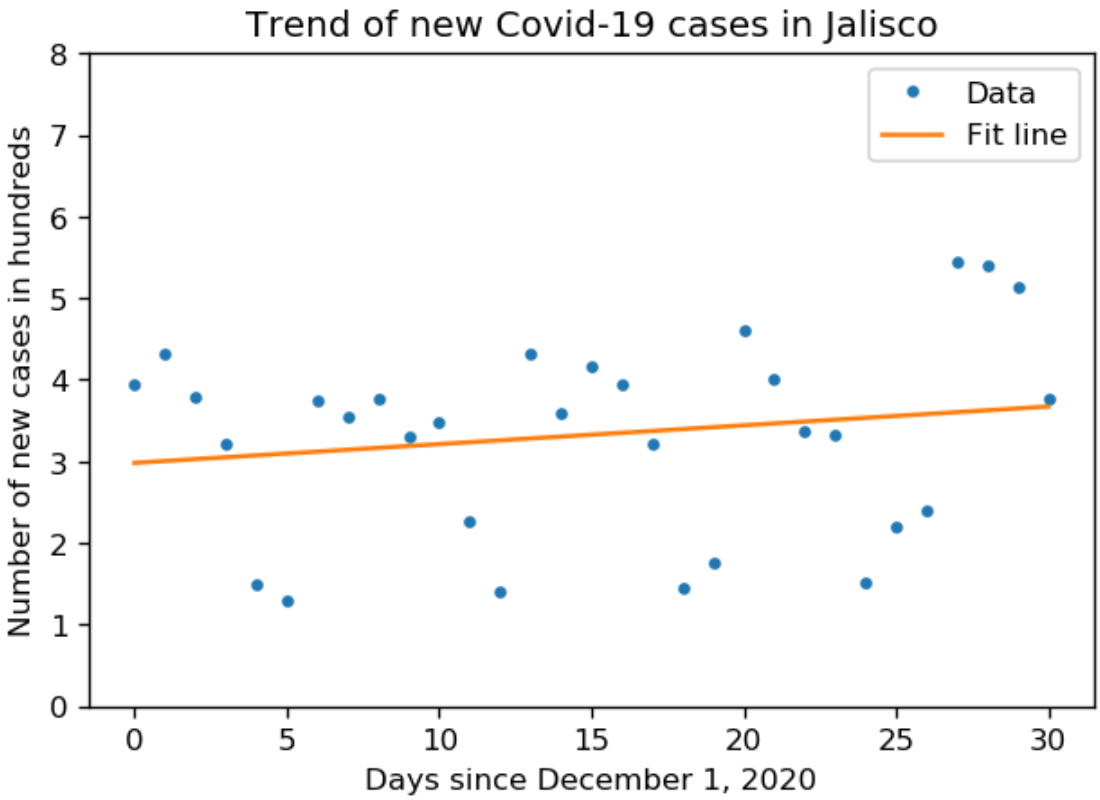
In [57]:
```
# BUILT IN NUMPY VERSION
x, sqrResidNorm, rank, sv = npla.lstsq( A, b, rcond=None )
print("x:", x)
```

```
x: [1236. 1943. 2416.]
```

In [58]:
```
# LEAST SQUARES LINEAR REGRESSION FITTING

# COVID CASE EXAMPLE

# Days from December 1, 2020.
t = np.array( range( 31 ) )

# Confirmed cases (in hundreds).
c = np.array( [3.94,4.32,3.80,3.21,1.50,1.29,3.74,3.54,3.77,3.30,3.47,2.27,
1.40,4.32,3.59,4.16,3.94,3.22,1.44,1.76,4.61,4.00,3.37,3.32,1.51,2.21,
2.40,5.44,5.41,5.14,3.76] )

# Create System
A = np.ones((len(t), 2))
A[:,1] = t
print("A shape:", A.shape)

# Solve for System
x = npla.lstsq( A, c, rcond=None )[0]
print("x:", x)

# Plot Graph
%matplotlib inline
plt.figure( dpi=120 )
plt.plot( t, c, ".", label="Data" )
lineT = np.linspace( 0, 30, num=4 ) #uses these points to plot the line
lineC = x[0] + x[1] * lineT
plt.plot( lineT, lineC, label="Fit line" )
plt.xlabel( r"Days since December 1, 2020" )
plt.ylabel( r"Number of new cases in hundreds" )
plt.ylim( [0, 8])
plt.title( r"Trend of new Covid-19 cases in Jalisco" )
plt.legend()
```

```
A shape: (31, 2)
x: [2.9822 0.023 ]
```

Out[58]:
```
<matplotlib.legend.Legend at 0x7f9c0e6c1250>
```



Rotations

Rotation Matrix: In 2D, $Ro(\alpha)@v$ rotates vector v by α degrees:

$$Ro(\alpha) = \begin{pmatrix} cos(\alpha) & sin(\alpha) \\ -sin(\alpha) & cos(\alpha) \end{pmatrix}$$

Reflection Matrix: To reflect vector v, multiply by:

$$Re = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

Reflection Matrix: To reflect vector v after rotating α degrees, multiply by:

$$Re(\alpha) = \begin{pmatrix} cos(\alpha) & sin(\alpha) \\ -sin(\alpha) & -cos(\alpha) \end{pmatrix}$$

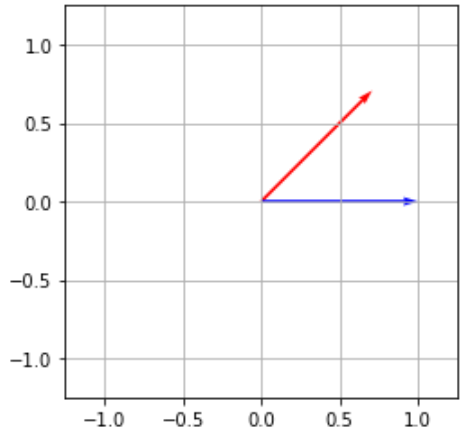
Rotation Transpose Property: $Ro(\alpha)^T = Ro(-\alpha)$
Rotation Addition Property: $Ro(\alpha_1)Ro(\alpha_2) = Ro(\alpha_1 + \alpha_2)$
Rotation Power Property: $Ro(\alpha)^n = Ro(n\alpha)$
Reflection Matrix Property:

$$Re(\alpha) = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} Ro(\alpha)$$

Informal Fact: If a function B from R to R^n satisfies the transpose and addition properties with $B(0) = I_n$, then $B(\alpha)$ is an orthogonal matrix.
Angle: The angle between two vectors is calculated using cosine similarity: $\alpha = cos^{-1} x^T y/||x||||y||$

In [59]:
```
def Ro(alpha):
    Rmm = np.array([[np.cos(alpha),np.sin(-alpha)],
                    [np.sin(alpha),np.cos(alpha)]])

    return Rmm

def angl(a,b):
    c = (a@b) / (npla.norm(a)*npla.norm(b))
    c = np.arccos(c)
    return np.degrees(c)
```

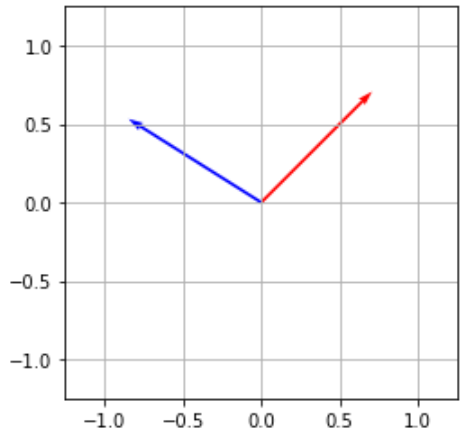
In [60]:
```
#unit vector at 45 degrees
v = np.array([1/np.sqrt(2),1/np.sqrt(2)])

#rotate -45 degrees
alpha = np.radians(-45)
v1 = Ro(alpha) @ v

plt.quiver([0,0],[0,0],[v[0],v1[0]],[v[1],v1[1]],angles='xy',scale_units='xy',scale=1
plt.xlim(-1.25, 1.25)
plt.ylim(-1.25, 1.25)
plt.grid(b=True, which='major');
plt.gca().set_aspect("equal")
plt.show()
print( "Norm of rotated vector:", npla.norm(v1), "\nAngle between vectors (degrees)",
```



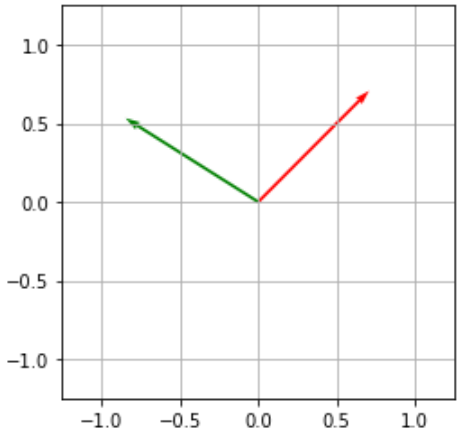

```
Norm of rotated vector: 0.9999999999999999
Angle between vectors (degrees) 45.0
```

In [61]:
```
#Transpose Property
A1 = np.round(Ro(alpha).T@Ro(alpha),1)
print("Is A1 the identity? ",np.all(A1==np.eye(2)))
print("The transpose is equal to an opposite rotation:",np.all(Ro(alpha).T==Ro(-alpha

#Addition Property
alpha1 = np.radians(147.8)
v1 = Ro(alpha1)@v1
plt.quiver([0,0],[0,0],[v[0],v1[0]],[v[1],v1[1]],angles='xy',scale_units='xy',scale=1
plt.xlim(-1.25, 1.25)
plt.ylim(-1.25, 1.25)
plt.grid(b=True, which='major');
plt.gca().set_aspect("equal")
plt.show()
print("Angle between vectors (degrees)", np.round(angl(v,v1),1))

v2 = Ro(alpha)@Ro(alpha1)@v
plt.quiver([0,0],[0,0],[v[0],v2[0]],[v[1],v2[1]],angles='xy',scale_units='xy',scale=1
plt.xlim(-1.25, 1.25)
plt.ylim(-1.25, 1.25)
plt.grid(b=True, which='major');
plt.gca().set_aspect("equal")
plt.show()
print("Angle between vectors (degrees)",np.round(angl(v,v2),1))

#Reflection Rotation Property
alpha = np.radians(103.4) #-75, 103.4
v1 = Ro(alpha)@v
Refl = np.array([[-1,0],[0,1]])@Ro(alpha) #Respect to x-axis, change be change to res
v2 = Refl@v
plt.quiver([0,0,0],[0,0,0],[v[0],v1[0],v2[0]],[v[1],v1[1],v2[1]],angles='xy',scale_ur
plt.xlim(-1.25, 1.25)
plt.ylim(-1.25, 1.25)
plt.grid(b=True, which='major');
plt.gca().set_aspect("equal")
plt.show()
print( "Norm under reflection:", np.round(npla.norm(v2),1), "\nAngle between vectors
```

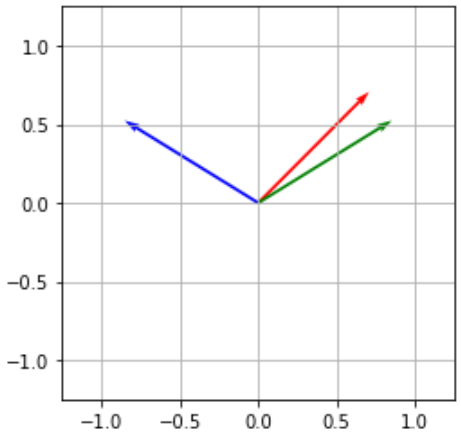
```
Is A1 the identity?  True
The transpose is equal to an opposite rotation: True
```




```
Angle between vectors (degrees) 102.8
```

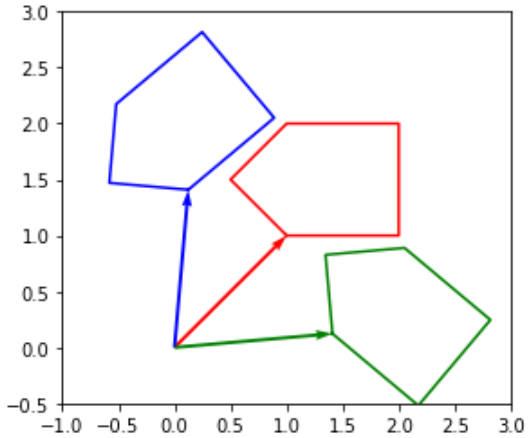



```
Angle between vectors (degrees) 102.8
```




```
Norm under reflection: 1.0
Angle between vectors (degrees) 13.4
```

In [62]:
```
# Orthogonal Transformation Preserves Length and Angle
# Norm = 1
x = [1,2,2,1,0.5,1]
y = [1,1,2,2,1.5,1]
plt.plot(x,y,color='r');
plt.quiver(0,0,x[0],y[0],angles='xy',scale_units='xy',scale=1,color='r')
plt.gca().set_aspect("equal")

alpha =  np.radians(40)
z = Ro(alpha)@np.array([x,y])
plt.plot(z[0,:],z[1,:],color='b');
plt.quiver(0,0,z[0,0],z[1,0],angles='xy',scale_units='xy',scale=1,color='b')


alpha =  np.radians(-40)
z = Ro(alpha)@np.array([x,y])
plt.plot(z[0,:],z[1,:],color='g');
plt.quiver(0,0,z[0,0],z[1,0],angles='xy',scale_units='xy',scale=1,color='g')

plt.gca().set_aspect("equal")

plt.xlim(-1,3)
plt.ylim(-0.5,3)

plt.show()
```



Singular Value Decomposition

Definition of SVD: Given a matrix A that is m x n, we can decompose into a product of 3 matrices, USV^T where U is m x m, S is m x n, and V is n x n.
Properties:

- $U^TU = I$; thus U is orthogonal (rotation/reflection)
- $V^TU = I$; thus V is orthogonal (rotation/reflection)
- S is a diagonal matrix (scaling)

Singular Values: non-zero elements in the diagonal matrix S where they are positive and appearing in (weak) descending order.
Note: Since $AV = US$ being m by n, think of A as a transformation function that maps from $R^n \rightarrow R^m$
Note: A performs a rotation/reflection, followed by stretch, folloed by another rotation/reflection
Note: U and V contain the bases for R^m and R^n respectively

Theorem 1: $Rank(A)$ is the number of nonzero singular values
Theorem 2: $||A||_2 = \sigma_0$: the two norm is the maximum singular value
Theorem 3:

$$\kappa_2(A) = \frac{\sigma_0}{\sigma_{min(m,n)-1}} = ||A||_2||A^{-1}||_2$$

Theorem 4: $||A||_F = \sqrt{\sum_i \sigma_i^2}$
Theorem 5: $|det(A)| = |\Pi_i \sigma_i|$

In [63]:
```
V = np.array([[12/13, 5/13], [-5/13, 12/13]])
print("V:", V, sep="\n")

U = np.array([[4/5, -3/5], [3/5, 4/5]])
print("U:", U, sep="\n")

sigma = [5, 1]
print("Sigmas:", sigma)

S = np.diag(sigma)
print("S:", S, sep="\n")

A = U @ S @ V.T
print("A:", A, sep="\n")

# ORTHOGONAL CHECK
print("Orthogonality Check:", npla.norm(U @ U.T - np.eye(2), 2))
print("Orthogonality Check:", npla.norm(U.T @ U - np.eye(2), 2))
print("Orthogonality Check:", npla.norm(V @ V.T - np.eye(2), 2))
print("Orthogonality Check:", npla.norm(V.T @ V - np.eye(2), 2))

# EQUALITY check
print("A @ V", A @ V, sep="\n")
print("U @ S", U @ S, sep="\n")
```

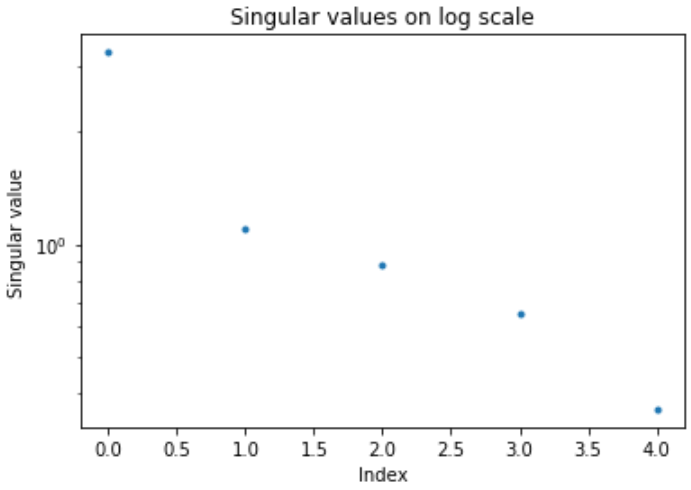
```
V:
[[ 0.9231  0.3846]
 [-0.3846  0.9231]]
U:
[[ 0.8 -0.6]
 [ 0.6  0.8]]
Sigmas: [5, 1]
S:
[[5 0]
 [0 1]]
A:
[[ 3.4615 -2.0923]
 [ 3.0769 -0.4154]]
Orthogonality Check: 2.6645352591003756e-17
Orthogonality Check: 2.6645352591003756e-17
Orthogonality Check: 2.220756981663096e-16
Orthogonality Check: 2.220756981663096e-16
A @ V
[[ 4.  -0.6]
 [ 3.   0.8]]
U @ S
[[ 4.  -0.6]
 [ 3.   0.8]]
```

In [64]:
```
# Singular Value Decomposition
A = np.random.rand(8, 5)
U, sigma, Vt = spla.svd(A)
S = np.zeros(A.shape)
for i in range(len(sigma)):
    S[i, i] = sigma[i]
V = Vt.T

print("A:", A, sep="\n")
print("U:", U, sep="\n")
print("sigma:", S, sep="\n")
print("V:", V, sep="\n")

# ORTHOGONAL CHECK
print("Orthogonality Check:", npla.norm(U.T @ U - np.eye(A.shape[0]), 2))
print("Orthogonality Check:", npla.norm(Vt.T @ Vt - np.eye(A.shape[1]), 2))

# EQUALITY CHECK
print("U @ S @ V.T:", npla.norm(U @ S @ Vt - A, 2))
```

```
A:
[[0.8325 0.7272 0.8167 0.6278 0.0401]
 [0.2783 0.6838 0.1315 0.7582 0.8844]
 [0.6132 0.3589 0.0327 0.1492 0.8297]
 [0.1478 0.2393 0.3218 0.8299 0.299 ]
 [0.753  0.9093 0.8638 0.1583 0.5835]
 [0.2629 0.8465 0.1873 0.12   0.5878]
 [0.5065 0.7012 0.8653 0.4286 0.5107]
 [0.2044 0.9264 0.4557 0.2265 0.0669]]
U:
[[-0.4199  0.4819  0.2696  0.261  -0.5894 -0.0344  0.0156 -0.319 ]
 [-0.3677 -0.5738  0.2897 -0.2178 -0.0369 -0.4895 -0.3438 -0.212 ]
 [-0.2662 -0.486  -0.284   0.4914 -0.3178 -0.0055  0.3509  0.3829]
 [-0.2307 -0.1187  0.6921 -0.0119  0.1599  0.5558  0.0146  0.3448]
 [-0.4667  0.195  -0.3845  0.2106  0.2795  0.1487 -0.6441  0.1921]
 [-0.2987 -0.2327 -0.3481 -0.4088 -0.0953  0.5616  0.1702 -0.4636]
 [-0.4172  0.1623  0.0319  0.1258  0.6316 -0.2538  0.5326 -0.1893]
 [-0.2889  0.2637 -0.1167 -0.6446 -0.1964 -0.2201  0.1753  0.5479]]
sigma:
[[3.2602 0.     0.     0.     0.    ]
 [0.     1.0969 0.     0.     0.    ]
 [0.     0.     0.8847 0.     0.    ]
 [0.     0.     0.     0.653  0.    ]
 [0.     0.     0.     0.     0.3621]
 [0.     0.     0.     0.     0.    ]
 [0.     0.     0.     0.     0.    ]
 [0.     0.     0.     0.     0.    ]]
V:
[[-0.4139  0.1346 -0.1757  0.6729 -0.5718]
 [-0.5965  0.0854 -0.3076 -0.6878 -0.263 ]
 [-0.4373  0.5921  0.0552  0.1795  0.6503]
 [-0.3458 -0.1562  0.9105 -0.0699 -0.1485]
 [-0.4025 -0.7743 -0.2058  0.1925  0.3988]]
Orthogonality Check: 1.380102136655585e-15
Orthogonality Check: 6.600899878666804e-16
U @ S @ V.T: 2.0668567819050656e-15
```

In [65]:
```
fig0 = plt.figure()
plt.plot( sigma, "." )
plt.yscale( "log" )
plt.title( "Singular values on log scale" )
plt.ylabel( "Singular value" )
plt.xlabel( "Index" )
```

Out[65]: Text(0.5, 0, 'Index')



In [66]:
```
# THEOREM 1 RANK(A) = # OF NONZERO VALUES
print( "Singular values:", sigma )
print( "rank( A ):", npla.matrix_rank( A ) )

B = A.copy()
B[:,4] = A @ [1, 1, 1, 1, 0]
# Can you see why this replaces column 4 of B with the sum of the first 4␣
#columns of A?
print( "B:" )
print( B )
print( "\nB @ [1, 1, 1, 1, -1] should be the zero vector:" )
print( B @ [1, 1, 1, 1, -1] )
print( "\nNorm of B @ [1, 1, 1, 1, -1]:", npla.norm( B @ [1, 1, 1, 1, -1] ) )


UB, sigmaB, VtB = spla.svd( B )

# machine determines singular values > machine epsilon (10^-16)
# to determine numerical rank since there is machine error
# exact rank of matrix is technically 5 since all values are nonzero
print( "Singular values of B:", sigmaB )
print("Rank(B):", npla.matrix_rank( B ))
```

```
Singular values: [3.2602 1.0969 0.8847 0.653  0.3621]
rank( A ): 5
B:
[[0.8325 0.7272 0.8167 0.6278 3.0042]
 [0.2783 0.6838 0.1315 0.7582 1.8518]
 [0.6132 0.3589 0.0327 0.1492 1.154 ]
 [0.1478 0.2393 0.3218 0.8299 1.5388]
 [0.753  0.9093 0.8638 0.1583 2.6843]
 [0.2629 0.8465 0.1873 0.12   1.4167]
 [0.5065 0.7012 0.8653 0.4286 2.5016]
 [0.2044 0.9264 0.4557 0.2265 1.813 ]]

B @ [1, 1, 1, 1, -1] should be the zero vector:
[0. 0. 0. 0. 0. 0. 0. 0.]

Norm of B @ [1, 1, 1, 1, -1]: 0.0
Singular values of B: [6.6299e+00 9.1085e-01 7.0608e-01 5.3608e-01 3.6733e-16]
Rank(B): 4
```

In [67]:
```
# THEOREM 2: NORM(A) is largest stretch of any vector
# corresponding to longest axis of ellipsoid, sigma_0

v0 = Vt.T[:,0]
u0 = U[:,0]
print( "Singular values of A:", sigma )
print( "2-norm of A:", npla.norm( A , 2 ) )
print( "\n v_0:", v0 )
print( " u_0:", u0 )
print( " A @ v_0:", A @ v0 )
print( "sigma_0 * u_0:", sigma[0] * u0 )
print( "\nnorm(A @ v_0) / norm(v_0):", npla.norm( A @ v0, 2 ) / npla.norm( v0, 2 ) )
```

```
Singular values of A: [3.2602 1.0969 0.8847 0.653  0.3621]
2-norm of A: 3.2601961429740696

 v_0: [-0.4139 -0.5965 -0.4373 -0.3458 -0.4025]
 u_0: [-0.4199 -0.3677 -0.2662 -0.2307 -0.4667 -0.2987 -0.4172 -0.2889]
 A @ v_0: [-1.3688 -1.1987 -0.8677 -0.752  -1.5214 -0.9738 -1.3601 -0.9418]
sigma_0 * u_0: [-1.3688 -1.1987 -0.8677 -0.752  -1.5214 -0.9738 -1.3601 -0.9418]

norm(A @ v_0) / norm(v_0): 3.2601961429740696
```

In [68]:
```
# THEOREM 3: The condition number is a ratio between largest
# and smallest singular values

print( "Ratio of extreme singular values:", sigma[0] / sigma[-1] )
print( "2-norm condition number of matrix:", npla.cond( A, 2 ) )
```

```
Ratio of extreme singular values: 9.004539984877312
2-norm condition number of matrix: 9.00453998487731
```

In [69]:
```
# THEOREM 4: Frobenius Norm equal to square root of
# sum of singular values / all values in matrix
print( "sqrt( sum( singular values squared ) ):", np.sqrt( np.sum( sigma ** 2 ) ) )
print( "sqrt( sum( matrix elements squared ) ):", np.sqrt( np.sum( A ** 2 ) ) )

print( "              Frobenius norm of matrix:", npla.norm( A, 'fro' ) )
```

```
sqrt( sum( singular values squared ) ): 3.629367421752507
sqrt( sum( matrix elements squared ) ): 3.629367421752508
              Frobenius norm of matrix: 3.629367421752508
```

In [70]:
```
# THEOREM 5: Determinant of Square Matrix is + or - of
# products of singular values
Asquare = A[:5,:]
print( "Shape of square matrix:", Asquare.shape )
UAs, sigmaAs, VtAs = npla.svd( Asquare )

print( "Product of singular values:", np.prod( sigmaAs ) )

print( "     Determinant of matrix:", npla.det( Asquare ) )
```

```
Shape of square matrix: (5, 5)
Product of singular values: 0.13428440392275437
     Determinant of matrix: -0.13428440392275431
```

Low Rank Approximation

Rank 1 Matrices: Let a and b be nonzero n-size vectors. The matrix $A = ab^T$ is n x n and Rank 1, since the columns of A are multiples of vector a with scalars from b.
Outer Product: ab^T which generates a matrix as opposed to $a^T b$ which generates a scalar.
Property: The product of two matrices is the sum of rank 1 matrices.
Theorem 6: Using SVD, $A = USV^T = \sum_{k=0}^{rank(A)-1} \sigma_k * u_k * v_k^T$ where u_k is a column in U and v_k is a column in V.
Theorem 7: Among all m-by-n matrices B_k that have rank k, the minimum possible value of $||A - B_k||_2$ is attained when $B_k = A_k$ as defined above. That value is $||A - A_k||_2 = \sigma_k$.
Theorem 8: Among all m-by-n matrices B_k that have rank k, the minimum possible value of $||A - B_k||_F$ is attained when $B_k = A_k$. That value is $||A - A_k||_F = \sqrt{\sum_{i \geq k} \sigma_i^2}$.

In [71]:
```
# THEOREM 6: Compute A from Rank 1 Matrices
Asum = np.zeros( A.shape )
for i in range( len( sigma ) ):
    Asum += sigma[i] * np.outer( U[:,i], V[:,i] )

print( "Norm of difference between Asum and A:",  npla.norm( Asum - A ) )
```

```
Norm of difference between Asum and A: 2.2298802502419194e-15
```

In [72]:
```
# THEOREM 7: Norm(A_k) = sigma_k

nrows, ncols = A.shape
print( "Shape of A:", (nrows, ncols) )

U, sigma, Vt = spla.svd( A )

print( "Singular values:", sigma )
print( "rank(A):", npla.matrix_rank( A ) )
print()

Ak = np.zeros( A.shape )
for k in range( len( sigma ) ):
    print( "Rank", npla.matrix_rank( Ak ),
           "approximation: 2-norm(A{} - A) = {}".format( k, npla.norm( Ak - A, 2 ) ) )
    Ak += sigma[k] * np.outer( U[:,k], Vt[k,:] )
print( "Rank", npla.matrix_rank( Ak ),
       "approximation: 2-norm(A{} - A) = {}".format( k + 1, npla.norm( Ak - A, 2 ) ) )
```

```
Shape of A: (8, 5)
Singular values: [3.2602 1.0969 0.8847 0.653  0.3621]
rank(A): 5

Rank 0 approximation: 2-norm(A0 - A) = 3.2601961429740696
Rank 1 approximation: 2-norm(A1 - A) = 1.096944771261387
Rank 2 approximation: 2-norm(A2 - A) = 0.8846890264189803
Rank 3 approximation: 2-norm(A3 - A) = 0.6529762984166883
Rank 4 approximation: 2-norm(A4 - A) = 0.36206137664438276
Rank 5 approximation: 2-norm(A5 - A) = 1.8758621206094133e-15
```

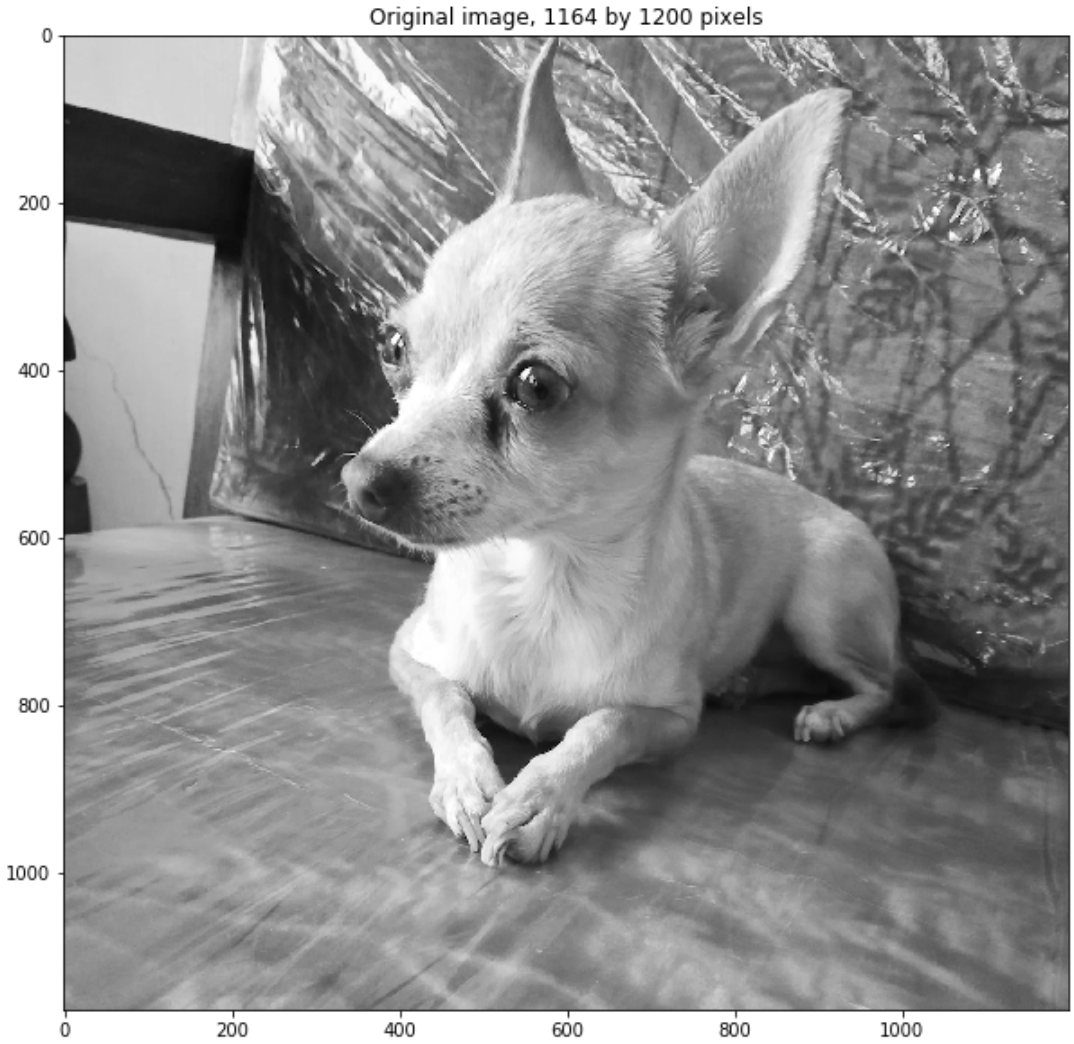
Image Compression

Lower Rank Approximation: using a Rank k approximation of A using a matrix U (m x k), S (k x k), and V.T (k x n).
Total Storage: (m + n + 1)k numbers stored. When k << m and n, a lot of storage can be saved.

In [73]:
```
# Read the image from a .jpg file and get just the black intensity of each pixel.
shiemi = plt.imread( "shiemi.jpg" )
M = np.float64( shiemi[:,:,0] )
nrows, ncols = M.shape
print( "Size of matrix M:", M.shape )

# Plot the original image (matrix).
plt.figure( figsize=(10, 10) )
plt.gray()
plt.imshow( M )
plt.title( "Original image, {} by {} pixels".format( nrows, ncols ) )
```

```
Size of matrix M: (1164, 1200)
```

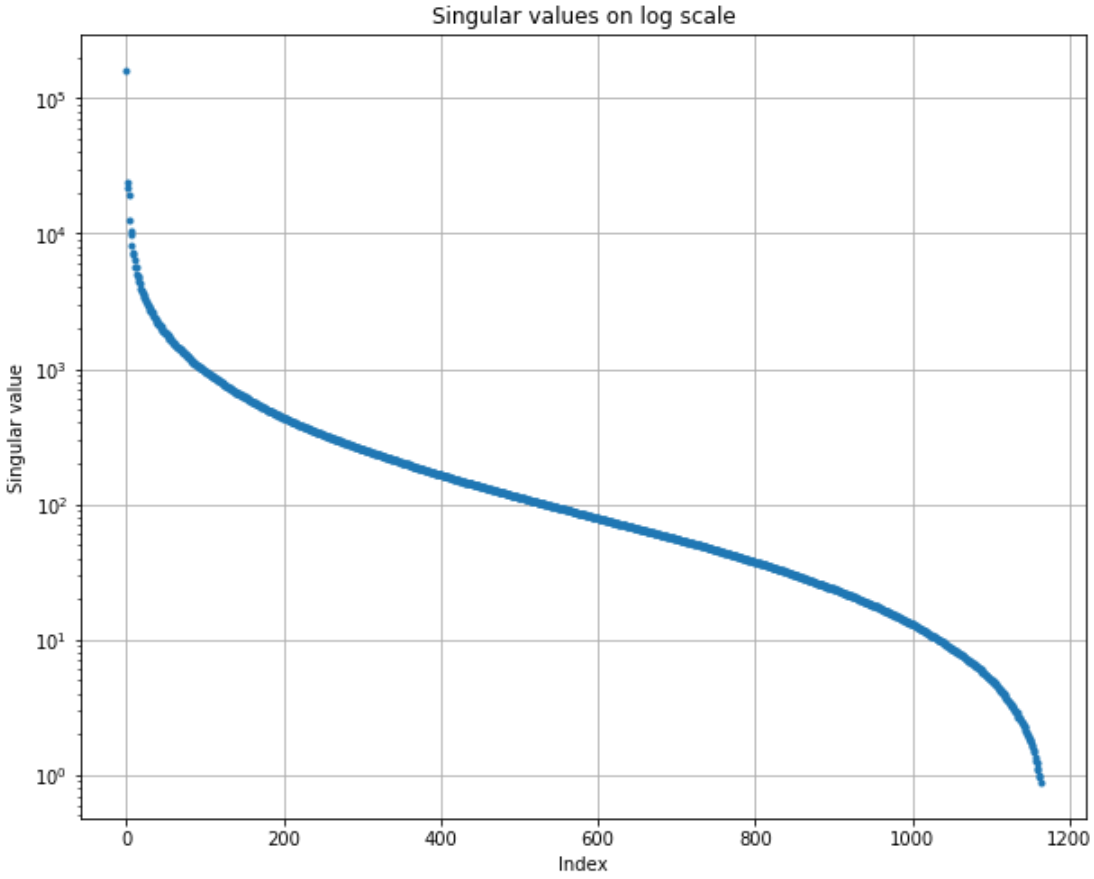
Out[73]:
```
Text(0.5, 1.0, 'Original image, 1164 by 1200 pixels')
```



In [74]:
```
nrows, ncols = M.shape
print( "Storage needed for all of M:", nrows * ncols )
print( "\nFirst 5 rows and cols of M:" )
print( M[:5,:5] )
```

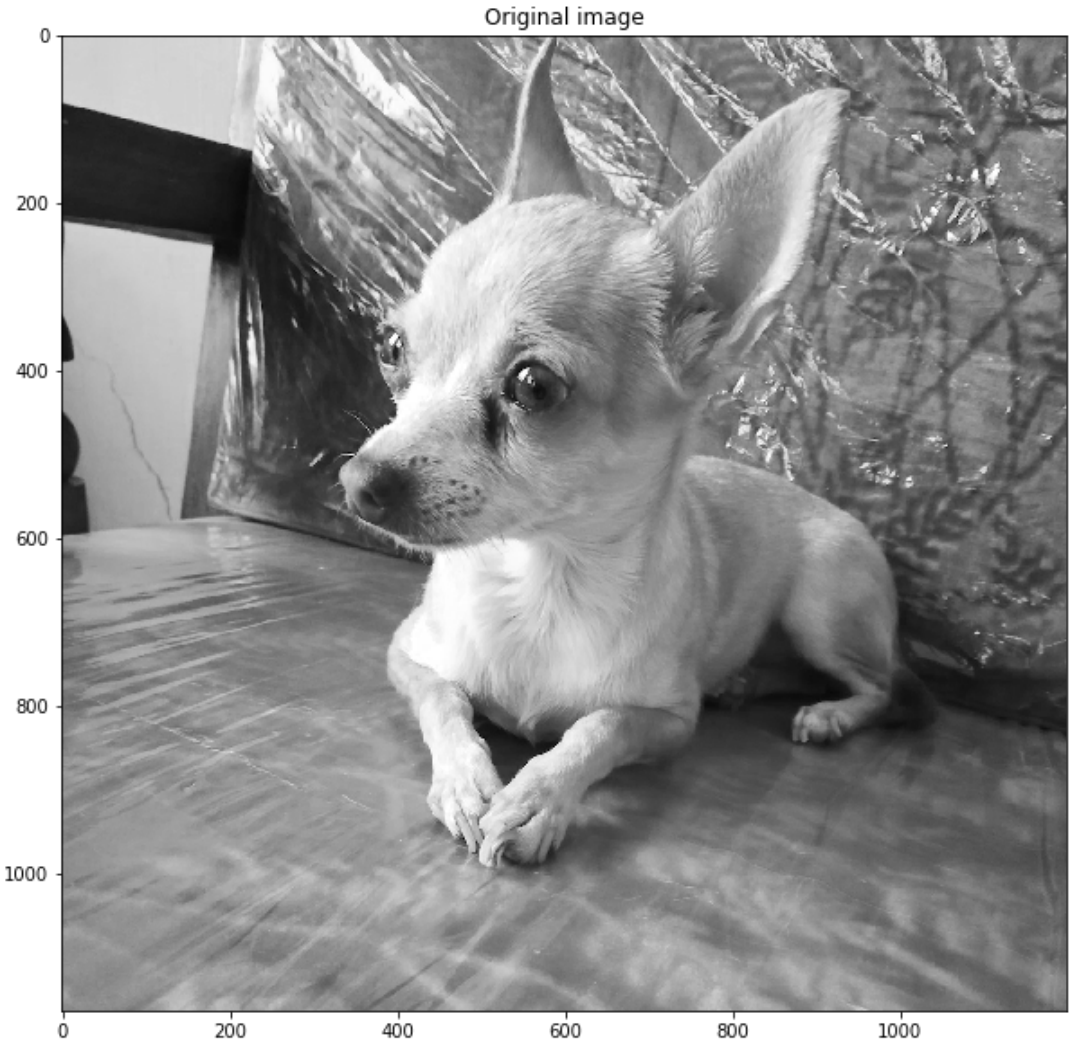
```
Storage needed for all of M: 1396800

First 5 rows and cols of M:
[[206. 205. 205. 203. 203.]
 [206. 205. 206. 204. 204.]
 [206. 205. 206. 205. 205.]
 [207. 206. 207. 206. 206.]
 [208. 207. 207. 206. 206.]]
```

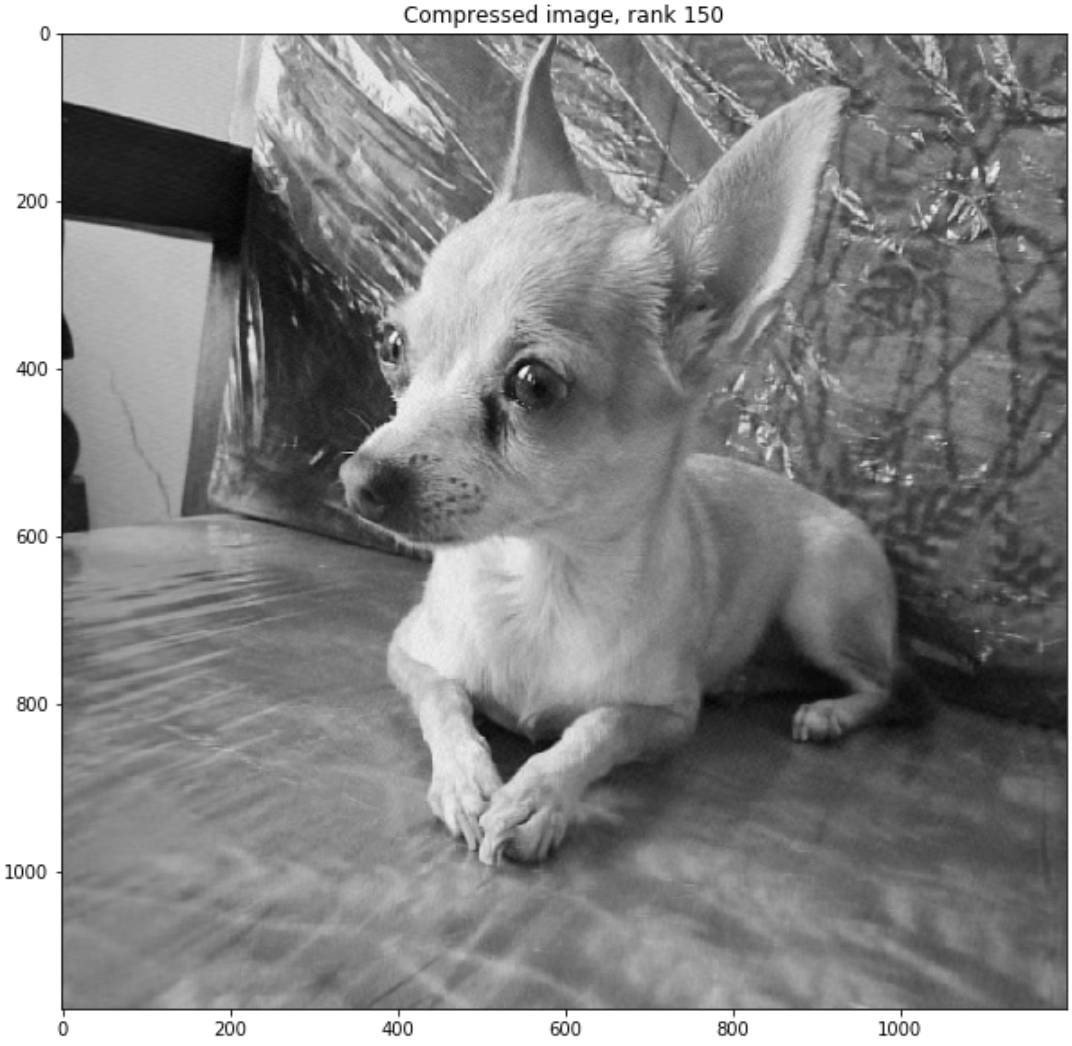
In [75]:
```
# SINGULAR VALUE DECOMPOSITION
U, sigma, Vt = spla.svd( M )

fig0 = plt.figure( figsize=(10,8) )
plt.plot( sigma, "." )
plt.yscale( "log" )
plt.title( "Singular values on log scale" )
plt.ylabel( "Singular value" )
plt.xlabel( "Index" )
plt.grid()
```



In [76]:
```
k = 150 # Try this first with 150, then with 50, then 10, then 2, then 1.

nrows, ncols = M.shape
Mk = np.zeros( M.shape )
for i in range( k ):
    Mk += sigma[i] * np.outer( U[:,i], Vt[i,:] )

print( "2-norm(M) =", npla.norm( M, 2 ) )
print( " sigma[0] =", sigma[0] )
print()
print( "2-norm(M{}-M) =".format( k ), npla.norm( Mk - M, 2 ) )
print( "    sigma[{}] =".format( k ), sigma[k] )
print()
print( "Relative error =", sigma[k] / sigma[0] )
print()

Mstorage = nrows * ncols
Mkstorage = k * (nrows + ncols + 1)

print( "Storage needed for all of M:", Mstorage )
print( "Storage needed for M{}:     ".format( k ), Mkstorage )
print()
print( "Compression factor:", Mstorage / Mkstorage )

# Plot the original image.
plt.figure( figsize=(10,10) )
plt.gray()
plt.imshow( M )
plt.title( "Original image" )
print()

# Plot the compressed image.
plt.figure( figsize=(10,10) )
plt.gray()
plt.imshow( Mk )
plt.title( "Compressed image, rank {}".format( k ) )
print()
```

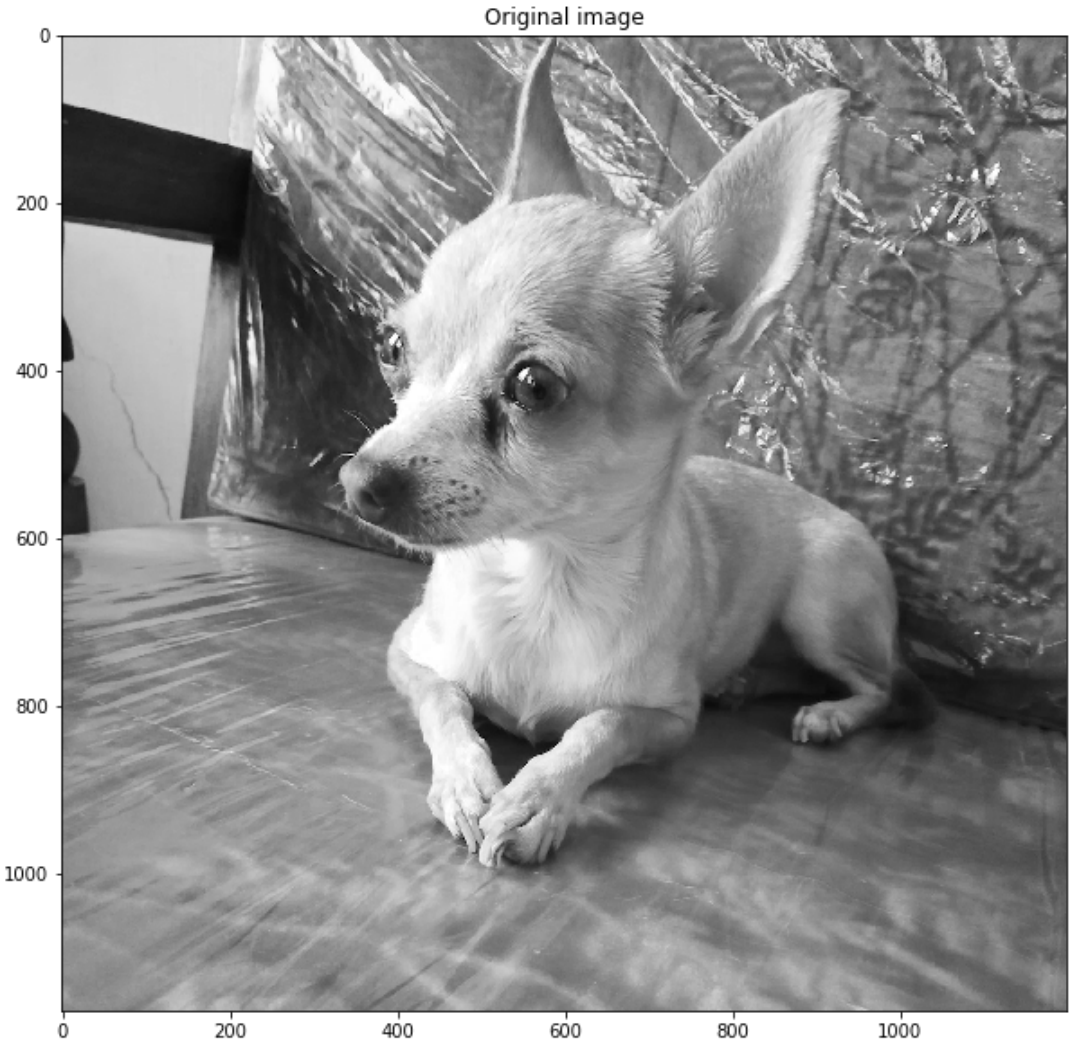
```
2-norm(M) = 161524.2427104553
 sigma[0] = 161524.24271045535

2-norm(M150-M) = 622.9593055082238
    sigma[150] = 622.9593055082237

Relative error = 0.0038567542249674943

Storage needed for all of M: 1396800
Storage needed for M150:      354750

Compression factor: 3.9374207188160675
```

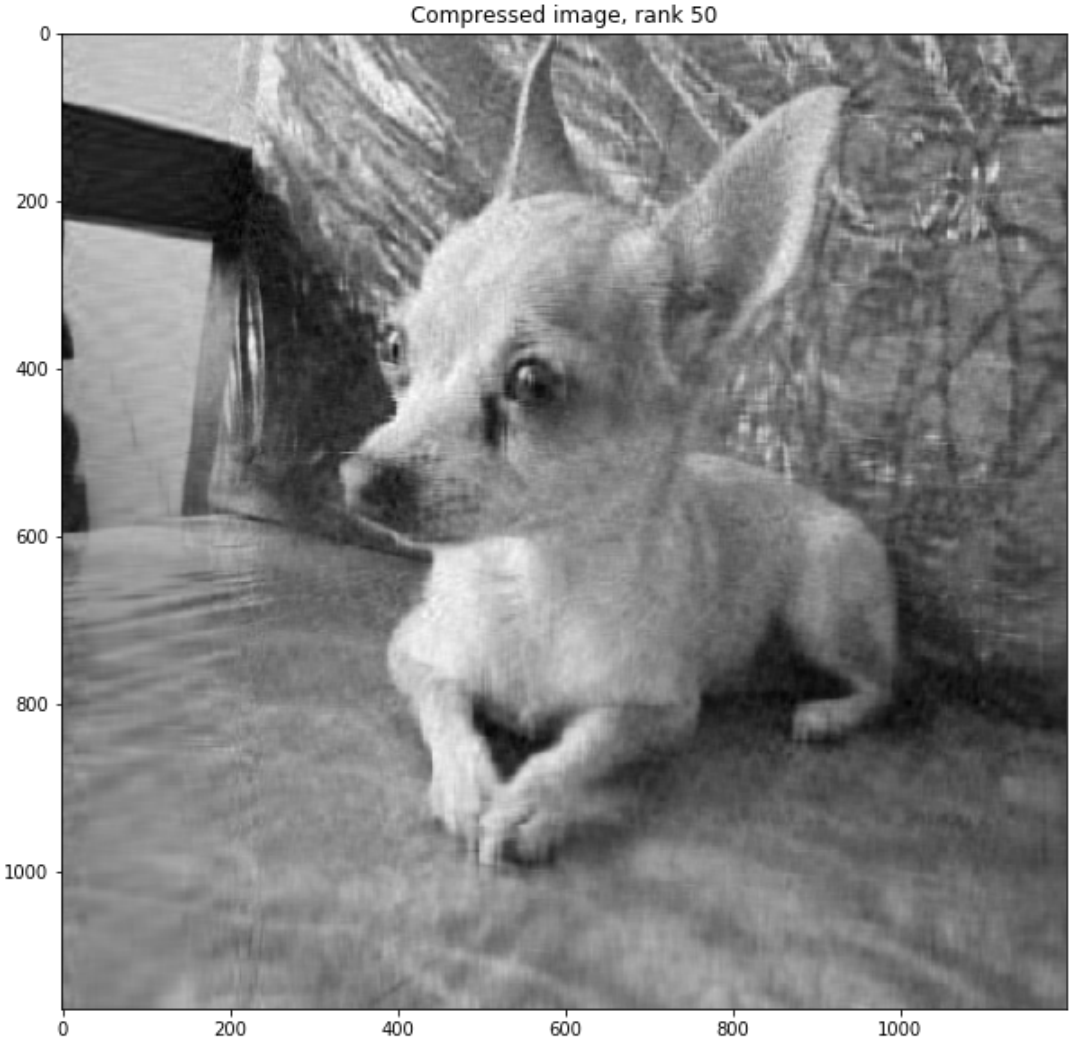

Original image
0
200
400
600
800
1000
0
200
400
600
800
1000


Compressed image, rank 150
0
200
400
600
800
1000
0
200
400
600
800
1000

In [77]:
```
k = 50 # Try this first with 150, then with 50, then 10, then 2, then 1.

nrows, ncols = M.shape
Mk = np.zeros( M.shape ) #start with zero matrix
for i in range( k ): #accumulate outer product
    Mk += sigma[i] * np.outer( U[:,i], Vt[i,:] )

print( "2-norm(M) =", npla.norm( M, 2 ) )
print( " sigma[0] =", sigma[0] )
print()
print( "2-norm(M{}-M) =".format( k ), npla.norm( Mk - M, 2 ) )
print( "    sigma[{}] =".format( k ), sigma[k] ) #by theorem
print()
print( "Relative error =", sigma[k] / sigma[0] ) #error
print()

Mstorage = nrows * ncols
Mkstorage = k * (nrows + ncols + 1)

print( "Storage needed for all of M:", Mstorage )
print( "Storage needed for M{}:     ".format( k ), Mkstorage )
print()
print( "Compression factor:", Mstorage / Mkstorage )

# Plot the original image.
plt.figure( figsize=(10,10) )
plt.gray()
plt.imshow( M )
plt.title( "Original image" )
print()

# Plot the compressed image.
plt.figure( figsize=(10,10) )
plt.gray()
plt.imshow( Mk )
plt.title( "Compressed image, rank {}".format( k ) )
print()
```

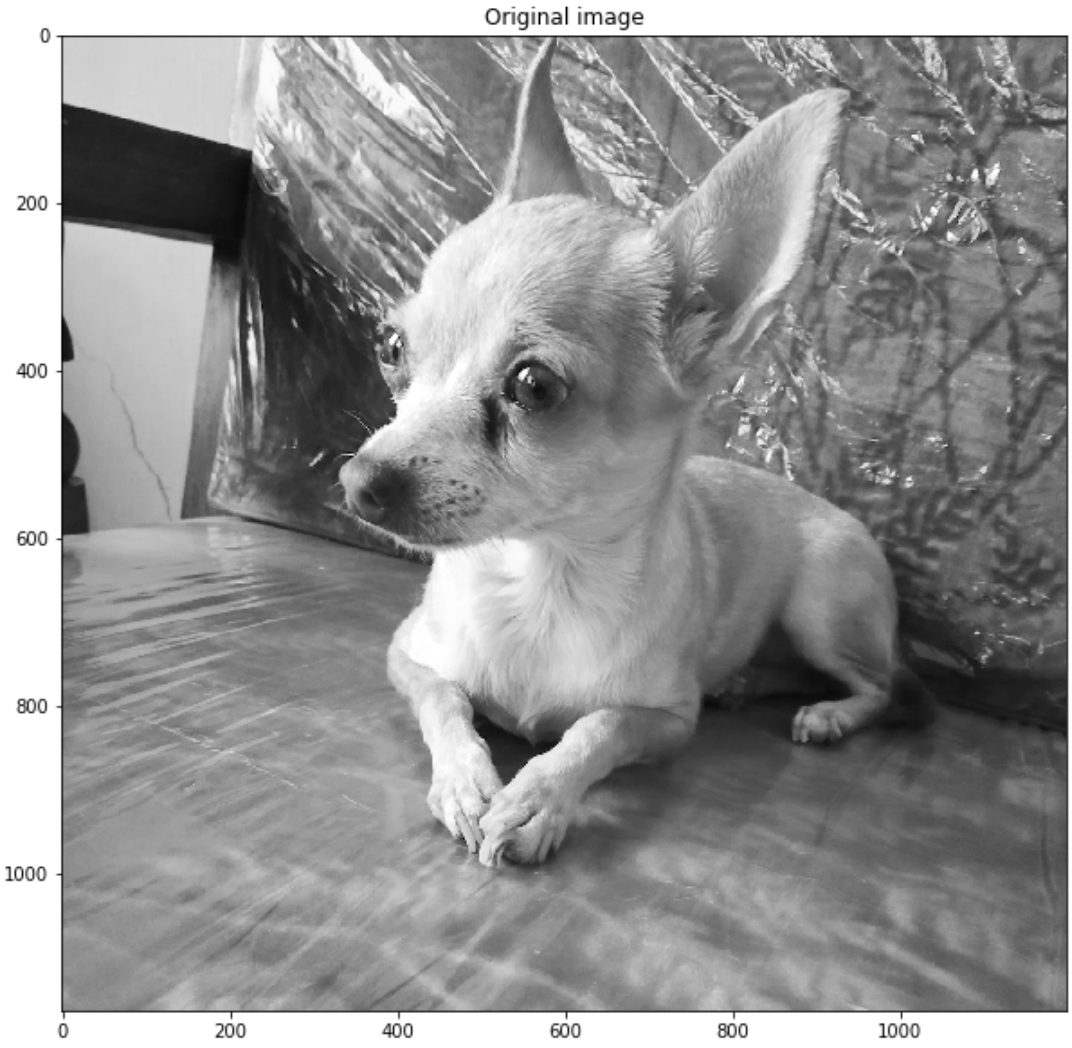
```
2-norm(M) = 161524.2427104553
 sigma[0] = 161524.24271045535

2-norm(M50-M) = 1829.6683809618587
    sigma[50] = 1829.6683809618607

Relative error = 0.011327515611645259

Storage needed for all of M: 1396800
Storage needed for M50:      118250

Compression factor: 11.812262156448202
```

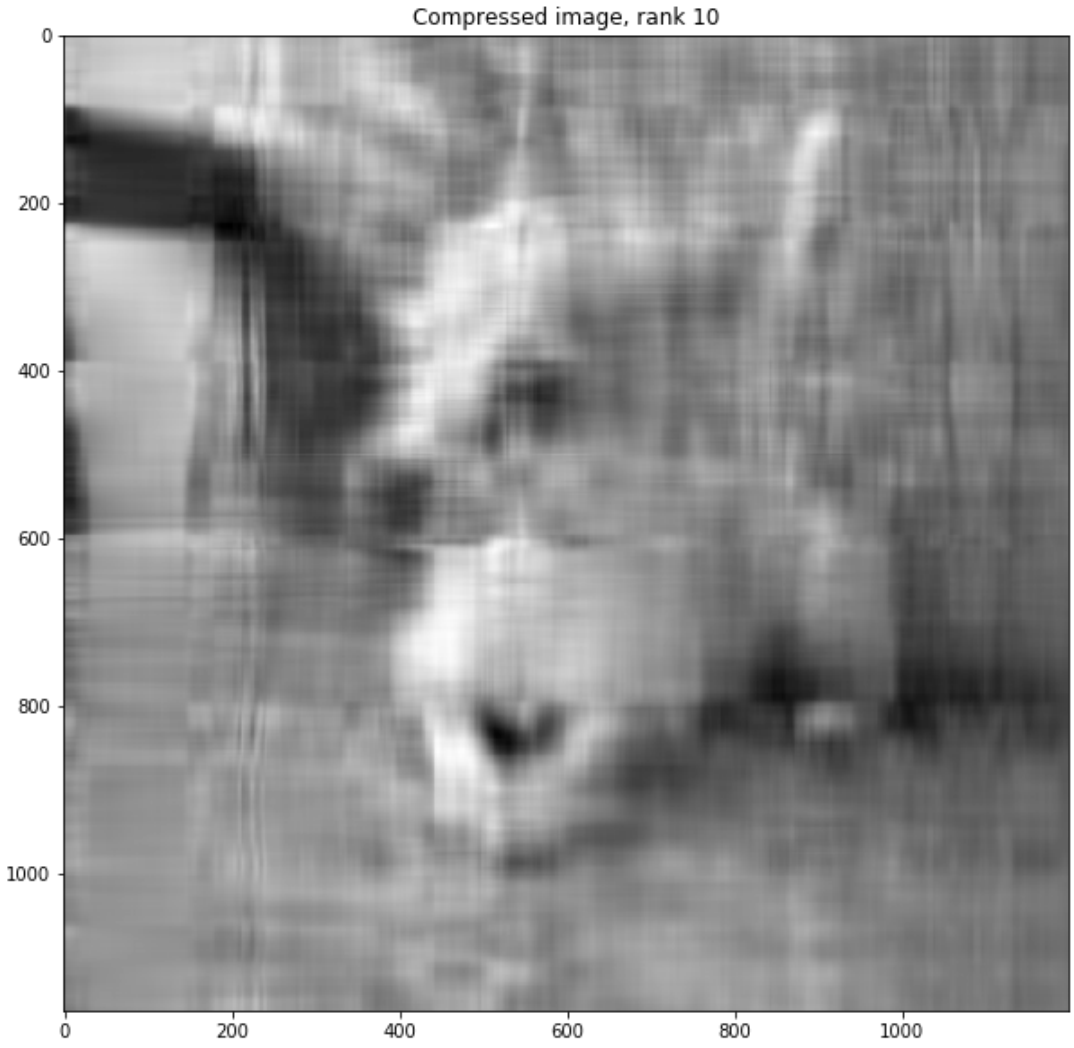

Original image
0
200
400
600
800
1000
0
200
400
600
800
1000


Compressed image, rank 50
0
200
400
600
800
1000
0
200
400
600
800
1000

In [78]:
```
k = 10 # Try this first with 150, then with 50, then 10, then 2, then 1.

nrows, ncols = M.shape
Mk = np.zeros( M.shape )
for i in range( k ):
    Mk += sigma[i] * np.outer( U[:,i], Vt[i,:] )

print( "2-norm(M) =", npla.norm( M, 2 ) )
print( " sigma[0] =", sigma[0] )
print()
print( "2-norm(M{}-M) =".format( k ), npla.norm( Mk - M, 2 ) )
print( "    sigma[{}] =".format( k ), sigma[k] )
print()
print( "Relative error =", sigma[k] / sigma[0] )
print()

Mstorage = nrows * ncols
Mkstorage = k * (nrows + ncols + 1)

print( "Storage needed for all of M:", Mstorage )
print( "Storage needed for M{}:     ".format( k ), Mkstorage )
print()
print( "Compression factor:", Mstorage / Mkstorage )

# Plot the original image.
plt.figure( figsize=(10,10) )
plt.gray()
plt.imshow( M )
plt.title( "Original image" )
print()

# Plot the compressed image.
plt.figure( figsize=(10,10) )
plt.gray()
plt.imshow( Mk )
plt.title( "Compressed image, rank {}".format( k ) )
print()
```

```
2-norm(M) = 161524.2427104553
 sigma[0] = 161524.24271045535

2-norm(M10-M) = 6486.6340471885505
    sigma[10] = 6486.634047188554

Relative error = 0.04015888846367381

Storage needed for all of M: 1396800
Storage needed for M10:      23650

Compression factor: 59.061310782241016
```


Original image
0
200
400
600
800
1000
0
200
400
600
800
1000



Eigenvalues and Eigenvectors

Definition: If $\mathbf{w}$ is a nonzero vector, and λ is a number, and $A\mathbf{w} = \lambda\mathbf{w}$, we say $\mathbf{w}$ is an *eigenvector* of A with *eigenvalue* λ. Notice that in this case any nonzero multiple of $\mathbf{w}$ is also an eigenvector.

Theorem: Every matrix has at least one eigenvalue/eigenvector, and an n-by-n matrix has at most n linearly independent eigenvectors.

Fact: The eigenvalues of A and A^T are the same, though the eigenvectors aren't necessarily the same.

Theorem: If A is an n-by-n symmetric matrix, then

- All the eigenvalues of A are real (no imaginary part),
- A has n linearly independent eigenvectors, and
- The eigenvectors can be chosen to be orthogonal to each other.

Eigendecomposition: $AW = WS$ holds where W is an orthogonal matrix ($W^TW = I$) and S is a square diagonal matrix. We can therefore write the eigenvalue equation as a matrix factorization: $A = WSW^T$

Note: S = diag($\lambda_0, \lambda_1, \ldots, \lambda_{n-1}$) with $\lambda_0 \leq \lambda_1 \leq \cdots \lambda_{n-1}$.

Note: the standard convention is to order eigenvalues in increasing order and singular values in decreasing order.

Note: $\mathbf{w}_i$ is column i of W. For all $0 \leq i < n$, $A\mathbf{w}_i = \lambda_i\mathbf{w}_i$

Note: for **symmetric matrices and their Eigendecomposition**, we can also refer to the eigenvectors individually as $\mathbf{q}_i$, and jointly as columns of Q since Q is orthogonal and the eigenvectors $\mathbf{q}_i$ are unit length and perpendicular to each other.

Definition: A symmetric matrix A is **positive definite** if all its eigenvalues are positive, so $0 < \lambda_0 \leq \lambda_1 \leq \cdots \lambda_{n-1}$.

Definition: A symmetric matrix A is **positive definite** if and only if $\mathbf{x}^T A\mathbf{x} > 0$ for all nonzero vectors $\mathbf{x}$.

Definition: A symmetric matrix A is **positive semidefinite** if all its eigenvalues are nonnegative, so $0 \leq \lambda_0 \leq \lambda_1 \leq \cdots \lambda_{n-1}$.

Definition: A symmetric matrix A is **positive semidefinite** if and only if $\mathbf{x}^T A\mathbf{x} \geq 0$ for all nonzero vectors $\mathbf{x}$.

Eigenvalue Shifting: Suppose A has eigenvalues $\lambda_0, \lambda_1, \ldots, \lambda_{n-1}$ with corresponding eigenvectors $\mathbf{w}_i$ such that $A\mathbf{w}_i = \lambda_i \mathbf{w}_i$.

Define $B = A - \alpha I$, then

$$\begin{aligned} B\mathbf{w} &= (A - \alpha I)\mathbf{w} \\ &= A\mathbf{w} - \alpha\mathbf{w} \\ &= \lambda\mathbf{w} - \alpha\mathbf{w} \\ &= (\lambda - \alpha)\mathbf{w} \end{aligned}$$

So, $\mathbf{w}$ is an eigenvector of B but with eigenvalue $\lambda - \alpha$.

In [79]:
```
# eigenvectors when transformed by matrix A are scaled by value lambda
# note: eigenvalues return complex form with real and complex part

# EXAMPLE 1: identity matrix (all vectors are eigenvectors w/ lambda=1)
A = np.eye( 3 )
print( "A:", A, sep="\n" )
evals, W = spla.eig( A )
print( "evals:", evals )
print( "W:", W, sep="\n" )

# EXAMPLE 2: 3 unique eigenvalues
# note: nonzero multiple of w is a vector for eignvalue lambda
A = np.diag( [1, 2, 3] )
print( "A:", A, sep="\n" )
evals, W = spla.eig( A )
print( "evals:", evals )
print( "W:", W, sep="\n" )

# EXAMPLE 3: eigenvalue = 0
# An eigenvalue can be zero (but an eigenvector can't be the zero vector).
A[1,1] = 0
print( "A:", A, sep="\n" )
evals, W = spla.eig( A )
print( "evals:", evals )
print( "W:", W, sep="\n" )

# EXAMPLE 4: permutation matrix (i.e., rotation)
# has complex eigen values (lambda = 1, -1, i, -i)
A = np.array( [[0,1,0,0], [0,0,1,0], [0,0,0,1], [1,0,0,0]] )
print( "A:", A, sep="\n" )
evals, W = spla.eig( A )
print( "evals:", evals )
print( "W:", W, sep="\n" )

# EXAMPLE 5: random
# note there are always complex conjugate pairs of eigenvalues
A = np.random.rand( 4, 4 )
print( "A:", A, sep="\n" )
evals, W = spla.eig( A )
print( "evals:", evals )
print( "W:", W, sep="\n" )

# Pull specific eigenvalue and eigenvector
i = 2         # Try real and complex evecs (0 and 2).
val = evals[i]
print("val:", val)
vec = W[:,i]
print("vec:", vec)
print("norm:", npla.norm( vec ))
print("val * vec:", val * vec)
print("A @ vec", A @ vec)
```

```
A:
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
evals: [1.+0.j 1.+0.j 1.+0.j]
W:
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
A:
[[1 0 0]
```

```
 [0 2 0]
 [0 0 3]]
evals: [1.+0.j 2.+0.j 3.+0.j]
W:
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
A:
[[1 0 0]
 [0 0 0]
 [0 0 3]]
evals: [1.+0.j 0.+0.j 3.+0.j]
W:
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
A:
[[0 1 0 0]
 [0 0 1 0]
 [0 0 0 1]
 [1 0 0 0]]
evals: [-1.0000e+00+0.j  8.3267e-17+1.j  8.3267e-17-1.j  1.0000e+00+0.j]
W:
[[-5.0000e-01+0.0000e+00j  1.3878e-16-5.0000e-01j  1.3878e-16+5.0000e-01j
  -5.0000e-01+0.0000e+00j]
 [ 5.0000e-01+0.0000e+00j  5.0000e-01+1.5266e-16j  5.0000e-01-1.5266e-16j
  -5.0000e-01+0.0000e+00j]
 [-5.0000e-01+0.0000e+00j -3.7470e-16+5.0000e-01j -3.7470e-16-5.0000e-01j
  -5.0000e-01+0.0000e+00j]
 [ 5.0000e-01+0.0000e+00j -5.0000e-01+0.0000e+00j -5.0000e-01-0.0000e+00j
  -5.0000e-01+0.0000e+00j]]
A:
[[0.407  0.545  0.823  0.5617]
 [0.6513 0.3166 0.8013 0.9169]
 [0.7896 0.5573 0.1523 0.407 ]
 [0.4691 0.8476 0.6234 0.8357]]
evals: [ 2.4376+0.j  0.2353+0.j -0.3814+0.j -0.5798+0.j]
W:
[[ 0.4655  0.4876 -0.4501 -0.4412]
 [ 0.5419 -0.083   0.7115 -0.4461]
 [ 0.3958  0.4576  0.2764  0.7754]
 [ 0.5771 -0.7389 -0.4635  0.0719]]
val: (-0.3814482651276098+0j)
vec: [-0.4501  0.7115  0.2764 -0.4635]
norm: 0.9999999999999999
val * vec: [ 0.1717-0.j -0.2714+0.j -0.1054+0.j  0.1768-0.j]
A @ vec [ 0.1717 -0.2714 -0.1054  0.1768]
```

In [80]:
```
#note: same eigenvalues, different eigenvectors

# A
evals, W = spla.eig( A )
print( "evals:", evals )
print( "W:" )
print( W )

# A.T
evals, W = spla.eig( A.T )
print( "evals:", evals )
print( "W:" )
print( W )
```

```
evals: [ 2.4376+0.j  0.2353+0.j -0.3814+0.j -0.5798+0.j]
W:
[[ 0.4655  0.4876 -0.4501 -0.4412]
 [ 0.5419 -0.083   0.7115 -0.4461]
 [ 0.3958  0.4576  0.2764  0.7754]
 [ 0.5771 -0.7389 -0.4635  0.0719]]
evals: [ 2.4376+0.j  0.2353+0.j -0.5798+0.j -0.3814+0.j]
W:
[[ 0.4715  0.6613 -0.4377  0.5494]
 [ 0.4737 -0.1402 -0.4228 -0.7421]
 [ 0.4888  0.3554  0.7597 -0.1473]
 [ 0.5607 -0.6455  0.2291  0.3547]]
```

In [81]:
```
# EIGENDECOMPOSITION

# Random symmetric matrix.
A = np.random.randn( 4, 4 )
A = A + A.T
print( "A:", A, sep="\n" )
evals, W = spla.eig( A )
print( "evals:", evals )
print( "W:", W, sep="\n" )

# When A is symmetric, we can use hermission and use more efficient algo.
# eigenvalues sorted in increasing order
evals, W = spla.eigh( A )
print( "evals:", evals )
print( "W:" )
print( W )

print("W.T @ W", W.T @ W, sep="\n")
S = np.diag( evals )
print("S", S, sep="\n")
print("W @ S @ W.T", W @ S @ W.T, sep="\n") #Confirm result
print("A", A, sep="\n") #Confirm result
```

```
A:
[[-2.2224 -0.3807  0.0179  0.3215]
 [-0.3807 -1.5847  1.3716 -1.3972]
 [ 0.0179  1.3716 -0.7055  0.2671]
 [ 0.3215 -1.3972  0.2671  0.6401]]
evals: [-3.0976+0.j -2.2293+0.j  1.449 +0.j  0.0053+0.j]
W:
[[-0.2379 -0.9634  0.1233  0.0071]
 [-0.7834  0.1327 -0.4948  0.352 ]
 [ 0.4852 -0.1406 -0.2106  0.8369]
 [-0.307   0.1856  0.8341  0.4191]]
evals: [-3.0976 -2.2293  0.0053  1.449 ]
W:
[[ 0.2379  0.9634  0.0071  0.1233]
 [ 0.7834 -0.1327  0.352  -0.4948]
 [-0.4852  0.1406  0.8369 -0.2106]
 [ 0.307  -0.1856  0.4191  0.8341]]
W.T @ W
[[ 1.0000e+00  1.1527e-16  2.2785e-16  1.8504e-16]
 [ 1.1527e-16  1.0000e+00 -7.2585e-17  5.0092e-17]
 [ 2.2785e-16 -7.2585e-17  1.0000e+00  1.4308e-16]
 [ 1.8504e-16  5.0092e-17  1.4308e-16  1.0000e+00]]
S
[[-3.0976  0.      0.      0.    ]
 [ 0.     -2.2293  0.      0.    ]
 [ 0.      0.      0.0053  0.    ]
 [ 0.      0.      0.      1.449 ]]
W @ S @ W.T
[[-2.2224 -0.3807  0.0179  0.3215]
 [-0.3807 -1.5847  1.3716 -1.3972]
 [ 0.0179  1.3716 -0.7055  0.2671]
 [ 0.3215 -1.3972  0.2671  0.6401]]
A
[[-2.2224 -0.3807  0.0179  0.3215]
 [-0.3807 -1.5847  1.3716 -1.3972]
 [ 0.0179  1.3716 -0.7055  0.2671]
 [ 0.3215 -1.3972  0.2671  0.6401]]
```

In [82]:
```
# Create an SPD matrix.
A = np.random.randn( 4, 4 )  # It's important that it's a random matrix!
A = A.T @ A
print( "A:", A, sep="\n" )
evals, W = spla.eigh( A )
print( "evals:", evals )
print( "W:" )
print( W )

# Make it semidefinite by shifting the eigenvalues by lambda_0.
B = A - evals[0] * np.eye( 4 )
print( "B:", B, sep="\n" )
evals, W = spla.eigh( B )
print( "evals:", evals )
print( "W:" )
print( W )

# The rank of a matrix is the number of nonzero eigenvalues.
print("Rank:", npla.matrix_rank( B ))

# w_0 is a nullvector of B.
print("B @ W[:,0]:", B @ W[:,0])
```

```
A:
[[ 0.6824  0.7234  1.3971  1.1052]
 [ 0.7234  6.8186 -4.2576  4.9642]
 [ 1.3971 -4.2576  9.0721 -0.285 ]
 [ 1.1052  4.9642 -0.285   5.7331]]
evals: [7.3909e-03 5.4700e-01 7.8462e+00 1.3906e+01]
W:
[[ 0.8022 -0.5389 -0.2569 -0.0077]
 [-0.449  -0.5418 -0.2454 -0.6669]
 [-0.3276 -0.1614 -0.7029  0.6104]
 [ 0.2181  0.6245 -0.6162 -0.4274]]
B:
[[ 0.675   0.7234  1.3971  1.1052]
 [ 0.7234  6.8112 -4.2576  4.9642]
 [ 1.3971 -4.2576  9.0647 -0.285 ]
 [ 1.1052  4.9642 -0.285   5.7257]]
evals: [-6.1581e-16  5.3961e-01  7.8388e+00  1.3898e+01]
W:
[[ 0.8022 -0.5389 -0.2569 -0.0077]
 [-0.449  -0.5418 -0.2454 -0.6669]
 [-0.3276 -0.1614 -0.7029  0.6104]
 [ 0.2181  0.6245 -0.6162 -0.4274]]
Rank: 3
B @ W[:,0]: [-6.1062e-16 -2.2204e-16  4.4409e-16  1.6653e-15]
```

Graphs and Laplacian Matrices

Definition: Let G be an undirected graph whose n vertices are the integers from 0 to $n-1$. The **Laplacian matrix** of G is the n-by-n matrix $L = L(G)$ whose entries are as follows:

- `L[i,i]` is the degree of vertex i (number of neighbors),
- `L[i,j] = -1` if (i,j) is an edge in G (and then also `L[j,i] = -1`), and
- The other elements of L are zero.

Theorem: Row sums of a graph is 0. Thus, the eigenvalue 0 exists for eigenvector containing only 1s.

Theorem: For any graph, L is positive semi-definite. That is, all eigenvalues of $L \geq 0$ and $x^T Lx \geq 0$ for all nonzero vectors x.

Fact: For any graph $G = (V, E)$, $L(G) = \sum_{(i,j) \in E} e_{ij} e_{ij}^T$ where e_{ij} is a vector following the rules of the Laplacian (contains a single -1, a single 1, and rest zeroes).

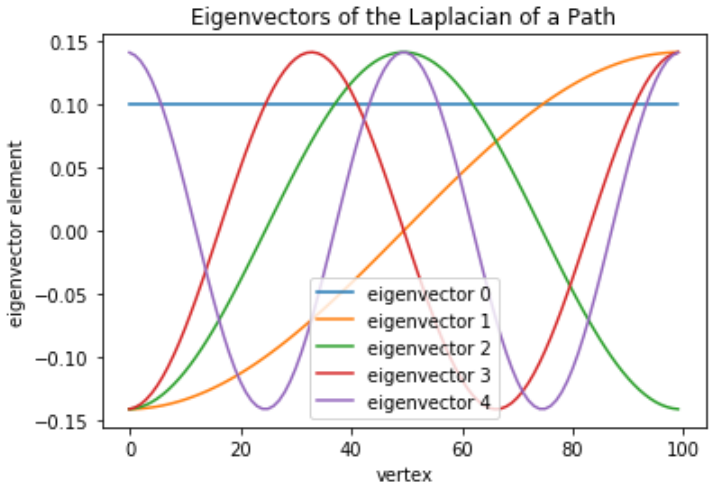
In [83]:
```
def path(n):
    """Laplacian matrix of the n-vertex path graph."""
    E = np.diag( np.ones( n - 1 ), -1 )
    L = 2 * np.eye( n ) - E - E.T
    L[0,0] = 1
    L[-1,-1] = 1
    return L
```

In [84]:
```
# EXAMPLE 1: SQUARE CYCLE
L = np.array( [[2, -1, 0, -1], [-1, 2, -1, 0], [0, -1, 2, -1], [-1, 0, -1, 2]] )
print("L:", L, sep="\n")
evals, Q = spla.eigh( L )
print( "evals:", evals )
print( "Q:", Q, sep="\n")

#PROPERTY: ROW SUMS ARE 0
print("Row Sums:", L @ np.ones( 4 ))

# EXAMPLE 2: PATH
L = path( 5 )
print("L:", L, sep="\n")
evals, Q = spla.eigh( L )
print( "evals:", evals )
print( "Q:", Q, sep="\n")

#PROPERTY: ROW SUMS ARE 0
print("Row Sums:", L @ np.ones( 5 ))
```

```
L:
[[ 2 -1  0 -1]
 [-1  2 -1  0]
 [ 0 -1  2 -1]
 [-1  0 -1  2]]
evals: [2.6645e-15 2.0000e+00 2.0000e+00 4.0000e+00]
Q:
[[-5.0000e-01  0.0000e+00  7.0711e-01  5.0000e-01]
 [-5.0000e-01 -7.0711e-01 -4.5756e-16 -5.0000e-01]
 [-5.0000e-01  2.8708e-16 -7.0711e-01  5.0000e-01]
 [-5.0000e-01  7.0711e-01 -1.7048e-16 -5.0000e-01]]
Row Sums: [0. 0. 0. 0.]
L:
[[ 1. -1.  0.  0.  0.]
 [-1.  2. -1.  0.  0.]
 [ 0. -1.  2. -1.  0.]
 [ 0.  0. -1.  2. -1.]
 [ 0.  0.  0. -1.  1.]]
evals: [0.    0.382 1.382 2.618 3.618]
Q:
[[ 4.4721e-01  6.0150e-01 -5.1167e-01  3.7175e-01 -1.9544e-01]
 [ 4.4721e-01  3.7175e-01  1.9544e-01 -6.0150e-01  5.1167e-01]
 [ 4.4721e-01  5.3654e-16  6.3246e-01  1.4024e-15 -6.3246e-01]
 [ 4.4721e-01 -3.7175e-01  1.9544e-01  6.0150e-01  5.1167e-01]
 [ 4.4721e-01 -6.0150e-01 -5.1167e-01 -3.7175e-01 -1.9544e-01]]
Row Sums: [0. 0. 0. 0. 0.]
```

In [85]:
```
# PATHS using different eigenvectors
# Note that the graph looks like increasing nodes of a wave
L = path( 100 )
evals, Q = spla.eigh( L )
plt.figure()

for i in range(5):
    plt.plot( Q[:,i], label="eigenvector {}".format( i ) )

plt.legend()
plt.xlabel( "vertex" )
plt.ylabel( "eigenvector element" )
plt.title( "Eigenvectors of the Laplacian of a Path" )
```

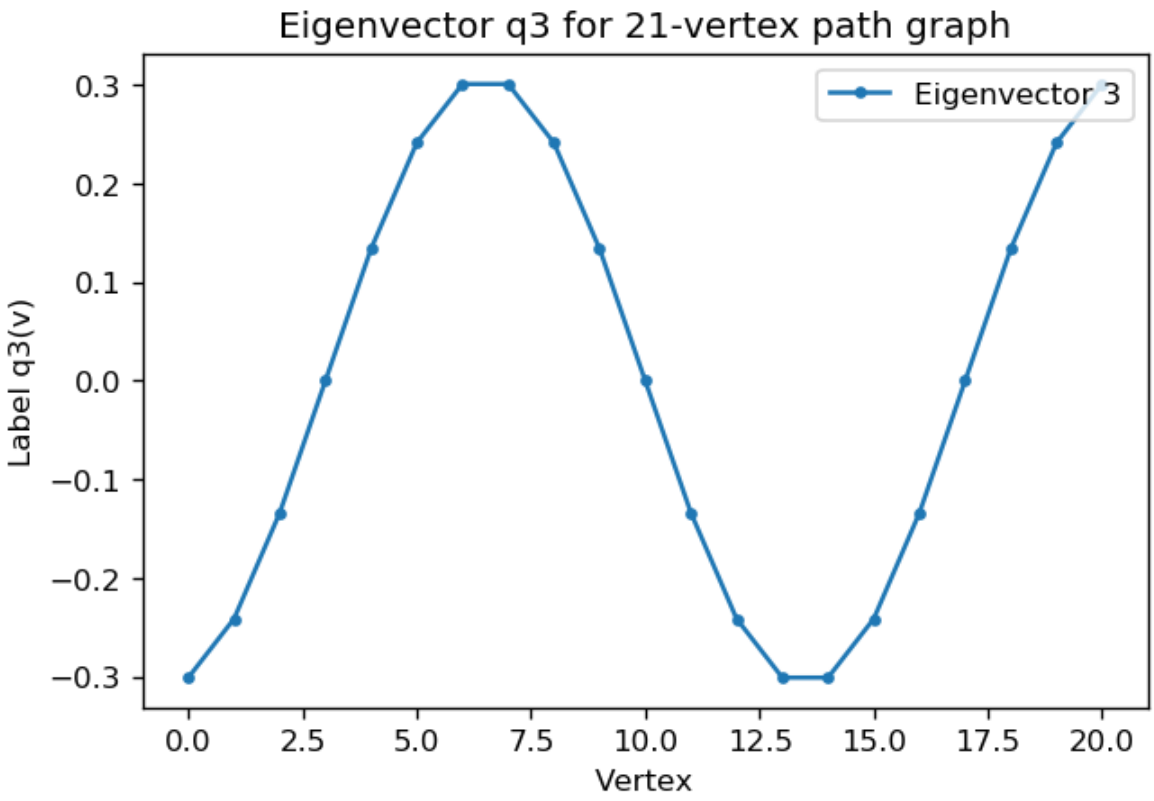
Out[85]:
```
Text(0.5, 1.0, 'Eigenvectors of the Laplacian of a Path')
```



In [86]:
```
L = cs111.path( 21 )
evals, Q = spla.eigh( L )
print( "Eigenvalues:", evals )

# Note labels are assigned to the vertices of x
i = 3
plt.figure( dpi=120 )
plt.plot( Q[:,i], '.-', label="Eigenvector {}".format( i ) )
plt.title( "Eigenvector q{} for {}-vertex path graph".format( i, Q.shape[0] ) )
plt.xlabel( "Vertex" )
plt.ylabel( "Label q{}(v)".format( i ) )
plt.legend()
```

```
Eigenvalues: [0.     0.0223 0.0889 0.1981 0.3475 0.5339 0.753  1.     1.2693 1.555
 1.8505 2.1495 2.445  2.7307 3.     3.247  3.4661 3.6525 3.8019 3.9111
 3.9777]
```

Out[86]:
```
<matplotlib.legend.Legend at 0x7f9bf88f8c50>
```



Laplacian Quadratic Form

Laplacian Quadratic Form (LQF): A scalar $x^T Lx = \sum_{(i,j)\in E}(x(i) - x(j))^2$.

Theorem: Since LQF is the sum of squares, L is SPSD.

Theorem: If the graph is connected, $x^T Lx = 0$ iff $x(i) = x(j)$ for all i and j.

In [87]:
```
# Theorem: L is symmetric positive semidefinite.
n = L.shape[0]
x = np.random.randn( n ) #generates from normal distribution
print( "Vector x:", x, sep="\n")
print( "x.T @ L @ x:", np.dot( x, L @ x ) )

# Theorem: the constant vector maps to a LQF of a 0 for a connected graph
n = L.shape[0]
x = np.ones( n )
print( "Vector x:", x, sep="\n")
print( "x.T @ L @ x:", np.dot( x, L @ x ) )
```

```
Vector x:
[ 0.3569  0.7237 -1.1926  0.0222 -0.179   0.1489  0.7607 -2.1845 -0.4883
  1.2117 -0.186  -0.4348  1.0848  0.362   0.571  -1.779  -0.2075 -0.7618
 -0.9423 -2.5152  0.2967]
x.T @ L @ x: 43.849288081630085
Vector x:
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
x.T @ L @ x: 0.0
```

Disconnected Graphs

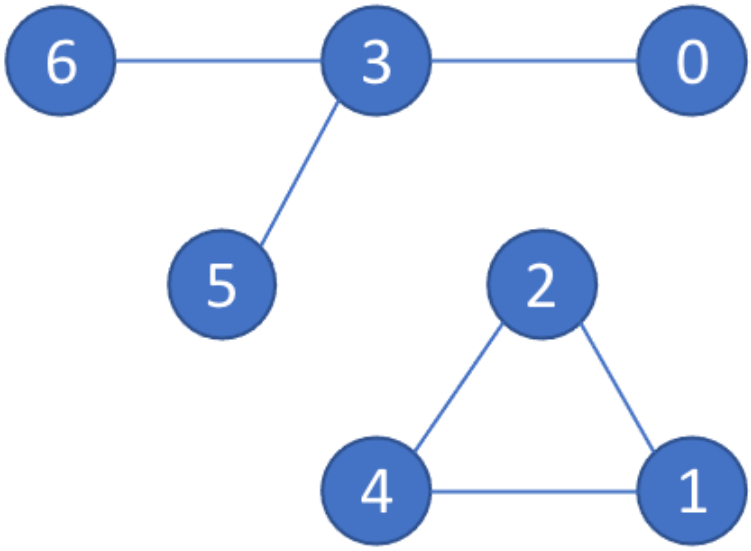
Theorem: If the graph is discoconnected, $x^T Lx = 0$ iff x is constant on every connected component of the graph.

Theorem: Suppose the eigenvalues of L(G) are $\lambda_0 \leq \ldots \leq \lambda_{n-1}$. Then,

- If G is connected: $0 = \lambda_0 < \lambda_1 \leq \ldots \lambda_{n-1}$
- If G is disconnected and has k connected components: $0 = \lambda_0 = \ldots = \lambda_{k-1} < \lambda_k \leq \ldots \lambda_{n-1}$

Note: The number of connected components of the graph is equal to the number of linearly independent null vectors.

Fiedler Value: λ_1 for any graph. As a result, when G is connected, it's the first nonzero eigenvalue. When G is not connected, the Fiedler Value is 0. Note that the **Fiedler Vector** is the associated eigenvector $q1$ such that $Lq_1 = \lambda_1 q_1$

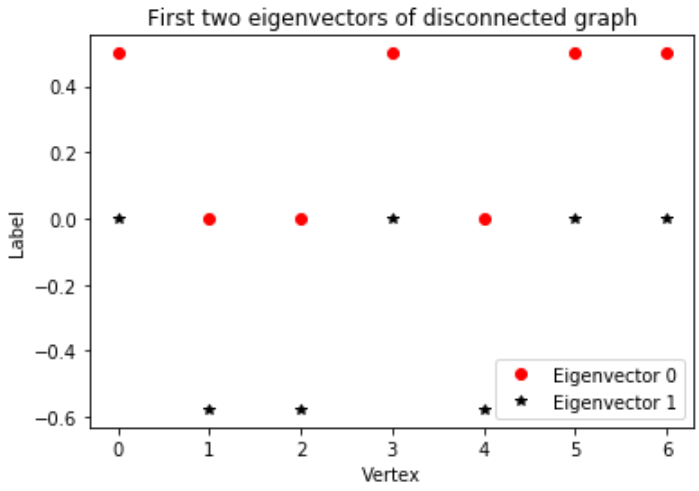

6
3
0
5
2
4
1

In [88]:
```
L = np.array( [[ 1,  0,  0, -1,  0,  0,  0,],
               [ 0,  2, -1,  0, -1,  0,  0],
               [ 0, -1,  2,  0, -1,  0,  0],
               [-1,  0,  0,  3,  0, -1, -1],
               [ 0, -1, -1,  0,  2,  0,  0],
               [ 0,  0,  0, -1,  0,  1,  0],
               [ 0,  0,  0, -1,  0,  0,  1]])
print("L:", L, sep="\n")

evals, Q = spla.eigh( L )
# 2 connected components = eigenvalue of 0 has multiplicity 2
print( "Eigenvalues:", evals )
print( "Q:", Q, sep="\n")
# lambda_1 = 0 for disconnected graph
print( "Fiedler value:", evals[1] )
# eigenvector_0 : always vector in null space
print( "Evec_0:", Q[:,0])
print( "L @ Evec_0:", L @ Q[:,0])
# eigenvector_1 : Fiedler vector (also in nullspace due to disconnected graph)
print( "Evec_1:", Q[:,1])
print( "L @ Evec_1:", L @ Q[:,1])
```

```
L:
[[ 1  0  0 -1  0  0  0]
 [ 0  2 -1  0 -1  0  0]
 [ 0 -1  2  0 -1  0  0]
 [-1  0  0  3  0 -1 -1]
 [ 0 -1 -1  0  2  0  0]
 [ 0  0  0 -1  0  1  0]
 [ 0  0  0 -1  0  0  1]]
Eigenvalues: [5.5511e-17 2.6645e-15 1.0000e+00 1.0000e+00 3.0000e+00 3.0000e+00
 4.0000e+00]
Q:
[[ 5.0000e-01  0.0000e+00 -8.1650e-01  0.0000e+00  0.0000e+00  0.0000e+00
  -2.8868e-01]
 [ 0.0000e+00 -5.7735e-01  0.0000e+00  0.0000e+00  0.0000e+00  8.1650e-01
   0.0000e+00]
 [ 0.0000e+00 -5.7735e-01  0.0000e+00 -5.5511e-17  7.0711e-01 -4.0825e-01
   0.0000e+00]
 [ 5.0000e-01  0.0000e+00 -3.6260e-16  0.0000e+00  0.0000e+00  0.0000e+00
   8.6603e-01]
 [ 0.0000e+00 -5.7735e-01  0.0000e+00  1.8953e-16 -7.0711e-01 -4.0825e-01
   0.0000e+00]
 [ 5.0000e-01 -5.5511e-17  4.0825e-01 -7.0711e-01 -1.1102e-16 -5.5511e-17
  -2.8868e-01]
 [ 5.0000e-01 -5.5511e-17  4.0825e-01  7.0711e-01  3.3307e-16 -5.5511e-17
  -2.8868e-01]]
Fiedler value: 2.6645352591003757e-15
Evec_0: [0.5 0.  0.  0.5 0.  0.5 0.5]
L @ Evec_0: [ 2.2204e-16  0.0000e+00  0.0000e+00 -3.3307e-16  0.0000e+00  0.0000e+0
0
  0.0000e+00]
Evec_1: [ 0.0000e+00 -5.7735e-01 -5.7735e-01  0.0000e+00 -5.7735e-01 -5.5511e-17
 -5.5511e-17]
L @ Evec_1: [ 0.0000e+00 -3.3307e-16  3.3307e-16  1.1102e-16  0.0000e+00 -5.5511e-1
7
 -5.5511e-17]
```

In [89]:
```
plt.figure()
plt.plot( Q[:,0], "ro", label = "Eigenvector 0" )
plt.plot( Q[:,1], "k*", label = "Eigenvector 1" )
plt.title( "First two eigenvectors of disconnected graph" )
plt.legend()
plt.xlabel( "Vertex" )
plt.ylabel( "Label" )
```

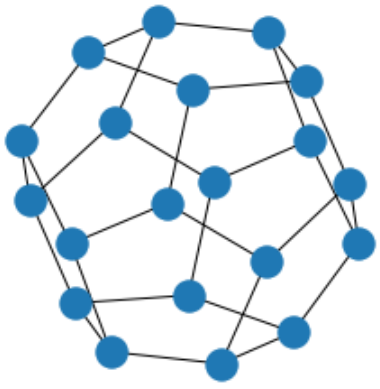
Out[89]:
```
Text(0, 0.5, 'Label')
```

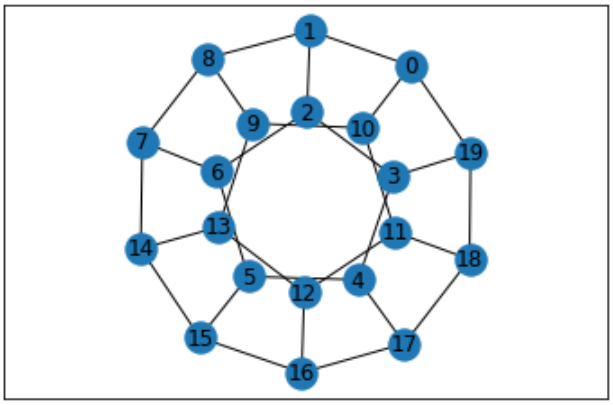


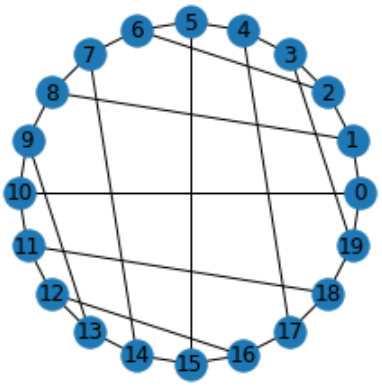
Embeddings

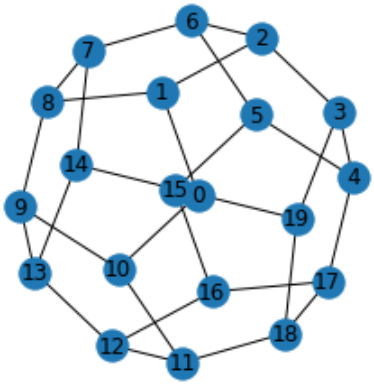
In [90]:
```
# Dodecahedron
G = nx.dodecahedral_graph()  # Platonic solid with 12 faces.
print("Edges:", G.number_of_edges())

plt.ion()
plt.figure()
plt.axis( "equal" )
nx.draw( G, pos=nx.spring_layout( G ) )  # Use spring layout.

plt.ion()
plt.figure()
plt.axis( "equal" )
nx.draw_networkx( G ) # Draw labelled network

plt.ion()
plt.figure()
plt.axis( "equal" )
nx.draw_circular( G, with_labels=True ) # Draw Circular Network

plt.ion()
plt.figure()
plt.axis( "equal" )
nx.draw_kamada_kawai( G, with_labels=True )

plt.ion()
plt.figure()
plt.axis( "equal" )
nx.draw_random( G, with_labels=True )
```

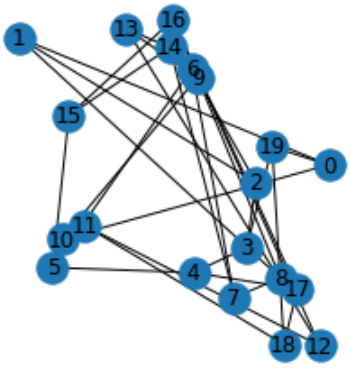
```
Edges: 30

/Library/anaconda3/lib/python3.7/site-packages/networkx/drawing/nx_pylab.py:579: Ma
tplotlibDeprecationWarning:
The iterable function was deprecated in Matplotlib 3.1 and will be removed in 3.3.
Use np.iterable instead.
  if not cb.iterable(width):
```




1
8
0
2
9
10
7
19
6
3
13
11
14
18
5
12
4
15
17
16


6
5
4
7
3
8
2
9
1
10
0
11
19
12
18
13
17
14
15
16


6
2
7
1
8
5
3
14
4
15
0
9
19
13
10
16
17
12
18
11


16
1
13
14
6
9
15
19
0
2
11
10
3
5
4
8
17
7
18
12

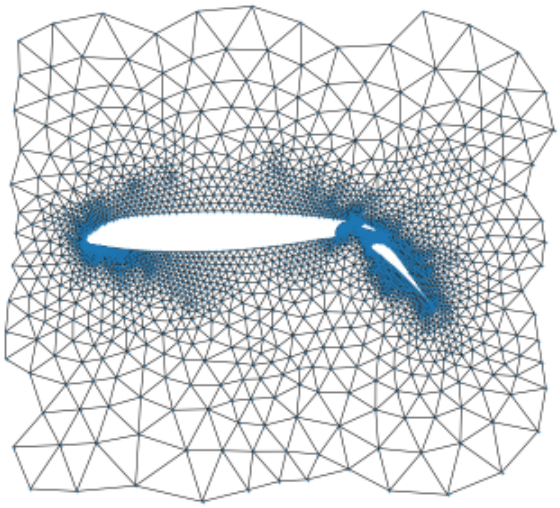
In [91]:
```
# Spectral Drawings

# Here are some of the graphs that networkx knows how to make. Try them all...

# G = nx.dodecahedral_graph()
# G = nx.house_graph()
# G = nx.moebius_kantor_graph()
# G = nx.petersen_graph()
G = nx.tutte_graph()  # See https://mathworld.wolfram.com/TuttesGraph.html.
# G = nx.truncated_tetrahedron_graph()
print( "Vertices:", G.nodes())
print( )
print( "Edges:", G.edges() )
print( )

L = nx.linalg.laplacian_matrix( G ).toarray()  # See all we need to do to retrieve F
print("Shape:", L.shape)

%time evals, Q = spla.eigh( L ) #evaluation time

print( "Q.shape:", Q.shape )
print( "First eigenvalues:", evals[:6] )

fiedlerValue = evals[1]
print( "Fiedler Value:", fiedlerValue)

fiedlerVector = Q[:,1]
print( "Fiedler Vector:", fiedlerVector)

# Use the first two eigenvectors as coordinates on the vertices.
coords = {}
for i in range( G.number_of_nodes() ):
    coords[i] = (Q[i,1], Q[i,2])       # Create a tuple for each vertex.

# Draw the graph with the vertices at the eigenvector coordinates
# (also called spectral coordinates).
plt.ion()
plt.figure()
plt.axis( "equal" )
nx.draw( G, pos=coords, node_shape='.' )  # Node we explicity say which coords we wa
```

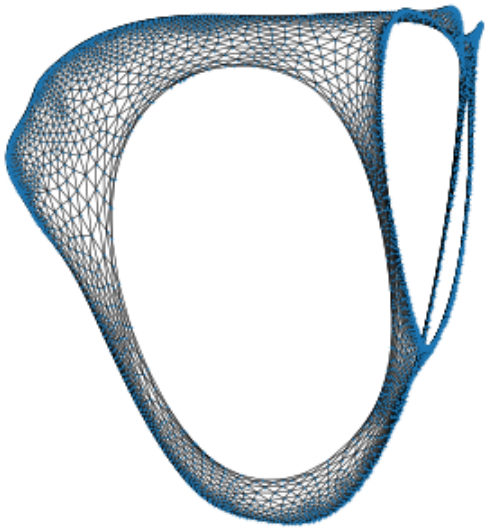
```
Vertices: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 2
0, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40,
41, 42, 43, 44, 45]

Edges: [(0, 1), (0, 2), (0, 3), (1, 4), (1, 26), (2, 10), (2, 11), (3, 18), (3, 1
9), (4, 5), (4, 33), (5, 6), (5, 29), (6, 7), (6, 27), (7, 8), (7, 14), (8, 9), (8,
38), (9, 10), (9, 37), (10, 39), (11, 12), (11, 39), (12, 13), (12, 35), (13, 14),
(13, 15), (14, 34), (15, 16), (15, 22), (16, 17), (16, 44), (17, 18), (17, 43), (1
8, 45), (19, 20), (19, 45), (20, 21), (20, 41), (21, 22), (21, 23), (22, 40), (23,
24), (23, 27), (24, 25), (24, 32), (25, 26), (25, 31), (26, 33), (27, 28), (28, 2
9), (28, 32), (29, 30), (30, 31), (30, 33), (31, 32), (34, 35), (34, 38), (35, 36),
(36, 37), (36, 39), (37, 38), (40, 41), (40, 44), (41, 42), (42, 43), (42, 45), (4
3, 44)]

Shape: (46, 46)
CPU times: user 1.59 ms, sys: 128 µs, total: 1.72 ms
Wall time: 1.03 ms
Q.shape: (46, 46)
First eigenvalues: [5.3291e-15 1.3646e-01 1.3646e-01 5.1459e-01 7.5392e-01 7.5392e-
01]
Fiedler Value: 0.13646258099316916
Fiedler Vector: [-8.0289e-16  1.4171e-01 -4.2230e-02 -9.9480e-02  1.9698e-01  1.948
```

```
2e-01
  1.3122e-01  2.0545e-02 -2.8010e-02 -4.9819e-02 -5.5323e-02 -6.5604e-02
 -7.9258e-02 -8.9101e-02 -4.4376e-02 -1.3151e-01 -1.7147e-01 -1.7887e-01
 -1.5349e-01 -1.3138e-01 -1.1556e-01 -4.2118e-02 -1.1601e-01  1.1096e-01
  1.9948e-01  2.2869e-01  2.0881e-01  1.6038e-01  2.1709e-01  2.2967e-01
  2.4577e-01  2.4656e-01  2.3158e-01  2.2754e-01 -5.8517e-02 -7.2254e-02
 -6.9128e-02 -5.9326e-02 -5.0934e-02 -6.6371e-02 -1.5857e-01 -1.5742e-01
 -1.7664e-01 -1.8723e-01 -1.8064e-01 -1.6117e-01]

/Library/anaconda3/lib/python3.7/site-packages/networkx/drawing/nx_pylab.py:579: Ma
tplotlibDeprecationWarning:
The iterable function was deprecated in Matplotlib 3.1 and will be removed in 3.3.
Use np.iterable instead.
  if not cb.iterable(width):
```

In [92]:
```
# Airfoils
airfoil, xycoords = cs111.read_mesh( "airfoil1" )
i= 0
for v in nx.nodes( airfoil ):
    print( type(v), ':', v, 'at', xycoords[v])
    i += 1
    if i > 10:
        break

plt.ion()
plt.figure( figsize=(6,6) )
plt.axis( "equal" )
nx.draw( airfoil, pos=xycoords, node_size=1, node_shape=".", width=0.5 )

L = nx.linalg.laplacian_matrix( airfoil ).toarray()
n = L.shape[0]
print("Shape:", n)

%time evals, Q = spla.eigh( L )

print( "Q.shape:", Q.shape )
print( "First eigenvalues:", evals[:6] )

coords = {}
for i in range(Q.shape[0]):
    coords[i] = (Q[i,1], Q[i,2])

plt.ion()
plt.figure( figsize=(6,6) )
plt.axis( "equal" )
nx.draw( airfoil, pos=coords, node_size=1, node_shape=".", width=0.5 )
```

```
<class 'int'> : 0 at (-0.132876, -0.338374)
<class 'int'> : 1 at (-0.132639, -0.264416)
<class 'int'> : 2 at (-0.124913, -0.180838)
<class 'int'> : 3 at (-0.118811, 0.134201)
<class 'int'> : 4 at (-0.115751, -0.084803)
<class 'int'> : 5 at (-0.112545, -0.576325)
<class 'int'> : 6 at (-0.109122, 0.334656)
<class 'int'> : 7 at (-0.106454, 0.001731)
<class 'int'> : 8 at (-0.098287, 0.522241)
<class 'int'> : 9 at (-0.096243, -0.123367)
<class 'int'> : 10 at (-0.096098, 0.058262)
Shape: 4253
CPU times: user 24.9 s, sys: 449 ms, total: 25.3 s
Wall time: 16.8 s
Q.shape: (4253, 4253)
First eigenvalues: [-2.0291e-15  1.8479e-03  4.4439e-03  6.2324e-03  8.7151e-03  1.
0360e-02]

/Library/anaconda3/lib/python3.7/site-packages/networkx/drawing/nx_pylab.py:579: Ma
tplotlibDeprecationWarning:
The iterable function was deprecated in Matplotlib 3.1 and will be removed in 3.3.
Use np.iterable instead.
  if not cb.iterable(width):
```





Covariance

Given a dataset $D \in R^{n*p}$,

- **Mean:** $\mu_x = n^{-1} \sum_{i=0}^{n-1} x_i$
- **Variance:** $\sigma_x^2 = (n-1)^{-1} \sum_{i=0}^{n-1} (x_i - \mu_x)^2$ with $n-1$ being defined as the **degrees of freedom**
- **Definition:** $r = x_i - \mu_x$. In vector form, $\sigma_x^2 = r^T r/(n-1)$
- **Note:** The variance is always nonnegative
- **Z-Score:** $\zeta_i = (x_i - \mu_x)/\sigma_x$

The covariance σ_{xy} of (discrete) variables x and y is defined as:

$$\operatorname{cov}(x, y) = \sigma_{xy} = \frac{1}{n-1}(\mathbf{x} - \mu_x)^T \left(\mathbf{y} - \mu_y\right),$$

- x and y **covary** if increments in x are generally accompanied by increments in y.
- x and y **anti-covary** if increments in one of them are accompanied by decrements in the other.
- x and y are **uncorrelated** if none of the above hold.

Let's now compute the **covariance** of $D \in \mathbb{R}^{n \times p}$. Define the p-element row vector μ^T, which contains the mean of each of the columns of D. Further, let M be the matrix with μ^T stacked n times. Then $(D - M)$ is closely related to D, except that each column (or variable) is *centered* around its mean. The matrix

$$C = \text{cov}(D) = \frac{1}{n-1}(D - M)^T(D - M)$$

is known as the **covariance matrix** of D, measuring how correlated (pairs of) columns are in the data set.

Now, if one lets $\mathbf{x}_i$ be the i^{th} column of $D - M$, then

$$\text{cov}(D) = \frac{1}{n-1}\begin{pmatrix} \mathbf{x}_0^T\mathbf{x}_0 & \mathbf{x}_0^T\mathbf{x}_1 & \cdots & \mathbf{x}_0^T\mathbf{x}_{p-1} \\ \mathbf{x}_1^T\mathbf{x}_0 & \mathbf{x}_1^T\mathbf{x}_1 & \cdots & \mathbf{x}_1^T\mathbf{x}_{p-1} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{x}_{p-1}^T\mathbf{x}_0 & \mathbf{x}_{p-1}^T\mathbf{x}_1 & \cdots & \mathbf{x}_{p-1}^T\mathbf{x}_{p-1} \end{pmatrix}.$$

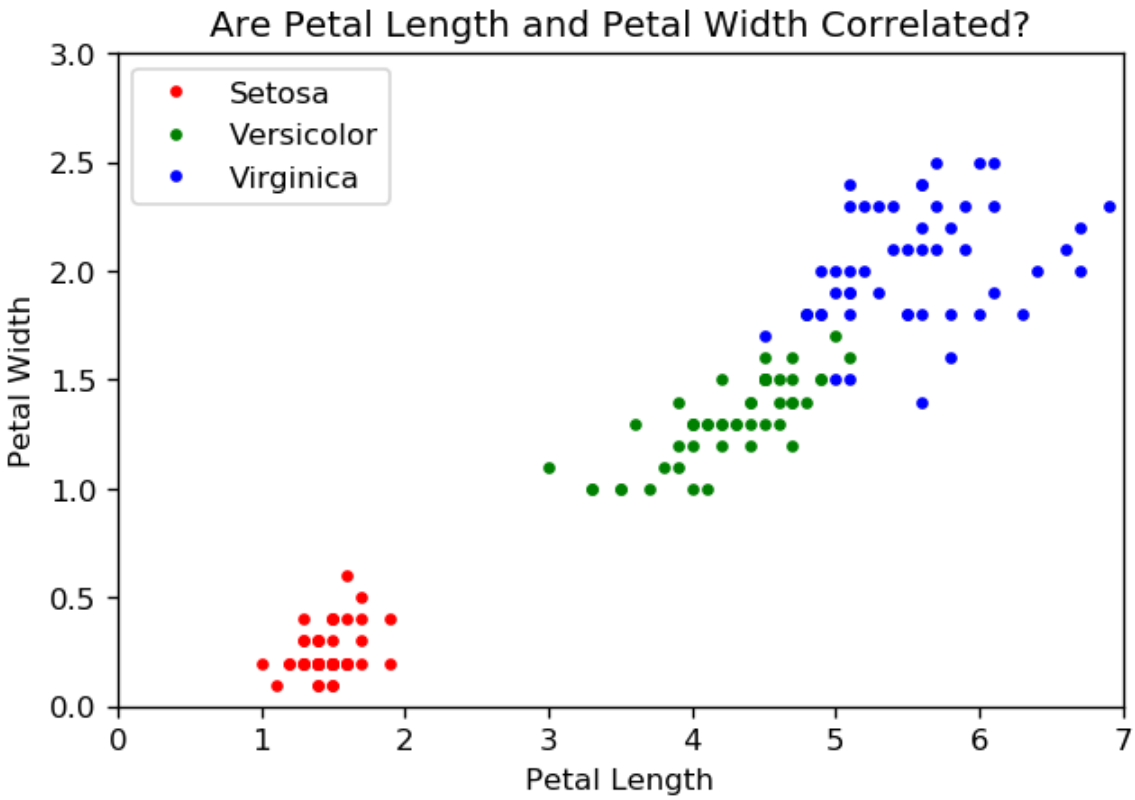
In particular, the diagonal elements in $C = \text{cov}(D)$ contain the variance of each of the columns in D.

Covariance Properties: Assuming that D has only real values:

- They are **symmetric positive semidefinite** (SPSD): their eigenvalues are all nonnegative.
- They are ellipsoids in p-dimensional space. Their eigenvectors determine the axes, and their eigenvalues determine the spread of the ellipsoids along these axes.
- Their eigendecomposition is similar to their singular value decomposition. That is, their eigenvalues and singular values are identical, and their corresponding eigen and singular vectors are equal up to a sign. In particular, $U = V$.

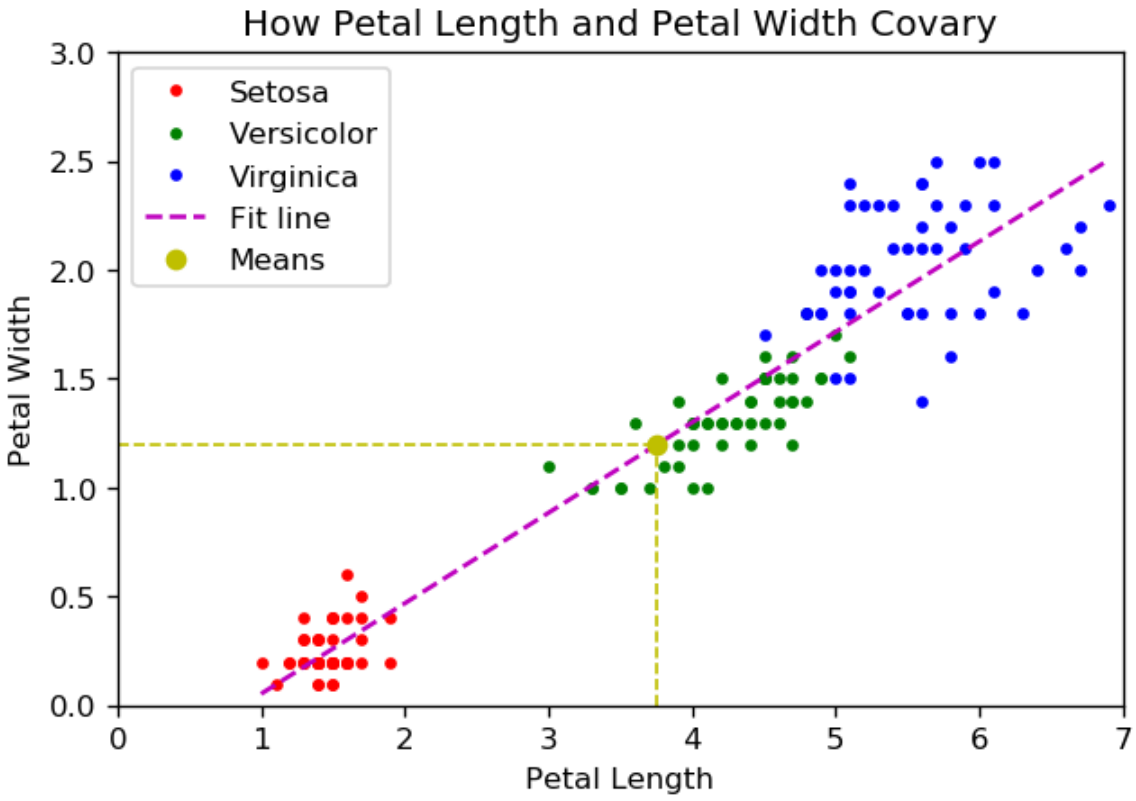
In [93]:
```
# Loading the iris data set.
from sklearn import datasets

iris = datasets.load_iris()

D = iris.data     # Contains features for 150 specimens (samples).
y = iris.target   # Contains the target class for each specimen.

(n, p) = D.shape
print( "Data set loaded: {} specimens and {} columns\n".format( n, p ) )

# Sneak peek at the contents of D and y.
print( "Sepal\tSepal\tPetal\tPetal\tIris" )
print( "Length\tWidth\tLength\tWidth\tSpecies" )
for i in range( 0, n, 15 ):
    for j in range( p ):
        print( D[i, j], end="\t" )
    print( y[i] )

# Let's investigate if **Petal Length** and **Petal Width** are correlated.

# Defining some indices.
setosaIdx = range( 0, 50 )          # The target classes appear in three consecutive r
versicolorIdx = range( 50, 100 )
virginicaIdx = range( 100, 150 )

# Plotting Petal Width against Petal Length.
plt.figure( dpi=120 )
plt.title( "Are Petal Length and Petal Width Correlated?" )
plt.plot( D[setosaIdx, 2], D[setosaIdx, 3], "r.", label="Setosa" )
plt.plot( D[versicolorIdx, 2], D[versicolorIdx, 3], "g.", label="Versicolor" )
plt.plot( D[virginicaIdx, 2], D[virginicaIdx, 3], "b.", label="Virginica" )
plt.xlabel( "Petal Length" )
plt.ylabel( "Petal Width" )
plt.legend()
plt.xlim( [0, 7] )
plt.ylim( [0, 3] )
plt.show()
```

```
Data set loaded: 150 specimens and 4 columns

Sepal   Sepal   Petal   Petal   Iris
Length  Width   Length  Width   Species
5.1     3.5     1.4     0.2     0
5.7     4.4     1.5     0.4     0
4.8     3.1     1.6     0.2     0
4.8     3.0     1.4     0.3     0
5.0     2.0     3.5     1.0     1
6.6     3.0     4.4     1.4     1
5.5     2.6     4.4     1.2     1
7.6     3.0     6.6     2.1     2
6.9     3.2     5.7     2.3     2
7.7     3.0     6.1     2.3     2
```


Are Petal Length and Petal Width Correlated?
Setosa
Versicolor
Virginica
Petal Width
Petal Length
0.0
0.5
1.0
1.5
2.0
2.5
3.0
0
1
2
3
4
5
6
7

In [94]:
```
# Let's compute the covariance of Petal Width and Petal Length.
x = D[:, 2]          # Petal length.
y = D[:, 3]          # Petal width.
mu_x = np.mean( x ) # Respective means.
mu_y = np.mean( y )

print( "Covariance of x and y is:", 1/(n - 1) * (x - mu_x).T @ (y - mu_y) )

# Let's find the regression line through the data, y = a + b*x, using least squares.
A = np.ones( (n, 2) )
A[:, 1] = x
Q, R = spla.qr( A, mode="economic" )   # Q is n-by-2 and R is 2-by-2.
coeff = cs111.Usolve( R, Q.T @ y )

# Plotting Petal Width against Petal Length.
plt.figure( dpi=120 )
plt.title( "How Petal Length and Petal Width Covary" )
plt.plot( x[setosaIdx], y[setosaIdx], "r.", label="Setosa" )
plt.plot( x[versicolorIdx], y[versicolorIdx], "g.", label="Versicolor" )
plt.plot( x[virginicaIdx], y[virginicaIdx], "b.", label="Virginica" )
x_min_max = np.array( [np.min( x ), np.max( x )] )
plt.plot( x_min_max, coeff[0] + coeff[1] * x_min_max, "m--", label="Fit line" )
plt.plot( mu_x, mu_y, "yo", label="Means" )
plt.plot( [mu_x, mu_x], [0, mu_y], "y--", linewidth=1 )
plt.plot( [0, mu_x], [mu_y, mu_y], "y--", linewidth=1 )
plt.xlabel( "Petal Length" )
plt.ylabel( "Petal Width" )
plt.legend()
plt.xlim( [0, 7] )
plt.ylim( [0, 3] )
plt.show()
```

```
Covariance of x and y is: 1.2956093959731547
```



In [95]:
```
# Computing the covariance matrix, step by step.
mu = np.mean( D, axis=0 )                       # 1-by-p row vector of column mean valu
print( "Mean row vector:" )
print( mu )

M = np.outer( np.ones( (n, 1) ), mu )           # n-by-p matrix of stacked row mean vec
C = 1 / (n - 1) * (D - M).T @ (D - M)           # p-by-p covariance matrix.

print( "\nCovariance matrix:" )
print( C )

C = np.cov( D.T )                               # Alternative method, but notice it nee
print( "\nCovariance matrix with alternative method:" )
print( C )

# Computing the variance of each of the columns in the data set.
print( "\nIndividual variances:" )
print( "Sepal length:", np.var( D[:,0], ddof=1 ) )  # Notice the ddof=1 parameter.
print( " Sepal width:", np.var( D[:,1], ddof=1 ) )
print( "Pepal length:", np.var( D[:,2], ddof=1 ) )
print( " Pepal width:", np.var( D[:,3], ddof=1 ) )
```

```
Mean row vector:
[5.8433 3.0573 3.758  1.1993]

Covariance matrix:
[[ 0.6857 -0.0424  1.2743  0.5163]
 [-0.0424  0.19   -0.3297 -0.1216]
 [ 1.2743 -0.3297  3.1163  1.2956]
 [ 0.5163 -0.1216  1.2956  0.581 ]]

Covariance matrix with alternative method:
[[ 0.6857 -0.0424  1.2743  0.5163]
 [-0.0424  0.19   -0.3297 -0.1216]
 [ 1.2743 -0.3297  3.1163  1.2956]
 [ 0.5163 -0.1216  1.2956  0.581 ]]

Individual variances:
Sepal length: 0.6856935123042507
 Sepal width: 0.189979418344519
Pepal length: 3.116277852348993
 Pepal width: 0.5810062639821029
```

In [96]:
```
# COVARIANCE PROPERTIES

# Eigendecomposition: SPSD, all nonnegative eigenvalues
evals, Q = spla.eigh( C )
print( "evals:", evals )
print( "evecs:" )
print( Q )

# SVD: U = V; matches eigenvalues and vectors(*-1)
U, sigma, Vt = spla.svd( C )
print( "sigma:", sigma )
print( "U:" )
print( U )
print( "V:" )
print( Vt.T )
```

```
evals: [0.0238 0.0782 0.2427 4.2282]
evecs:
[[ 0.3155 -0.582  -0.6566  0.3614]
 [-0.3197  0.5979 -0.7302 -0.0845]
 [-0.4798  0.0762  0.1734  0.8567]
 [ 0.7537  0.5458  0.0755  0.3583]]
sigma: [4.2282 0.2427 0.0782 0.0238]
U:
[[-0.3614 -0.6566  0.582   0.3155]
 [ 0.0845 -0.7302 -0.5979 -0.3197]
 [-0.8567  0.1734 -0.0762 -0.4798]
 [-0.3583  0.0755 -0.5458  0.7537]]
V:
[[-0.3614 -0.6566  0.582   0.3155]
 [ 0.0845 -0.7302 -0.5979 -0.3197]
 [-0.8567  0.1734 -0.0762 -0.4798]
 [-0.3583  0.0755 -0.5458  0.7537]]
```

Principal Component Analysis

Principal Component Analysis is based on the **singular value decomposition** of the covariance matrix C of a data set D. If $C = USV^T$, the columns of U (or V) are the **principal components** of D. If we take the k first principal components, we can get the approximation

$$\mathbf{d} \approx \mu + a_0\mathbf{e}_0 + a_1\mathbf{e}_1 + \cdots + a_{k-1}\mathbf{e}_{k-1},$$

where the coefficients $a_i = \mathbf{x} \cdot \mathbf{e}_i$ are the normal projections of the centered data row vectors $\mathbf{x} = \mathbf{d} - \mu$ onto the eigenvectors $\mathbf{e}_i$. Let $U^{(k)}$ be the p-by-k matrix of first k columns of U; then, we can write the previous approximation in matrix form for the whole data set D as

$$D^{(k)} \approx M + (XU^{(k)})\left(U^{(k)}\right)^T,$$

where $M \in \mathbb{R}^{n\times p}$ is the matrix with the mean row vector μ^T stacked n times and $X = D - M$ is the n-by-p matrix of centered data. Notice that we can retrieve the original data set $D = D^{(p)}$ if we use all of the p singular vectors in U.

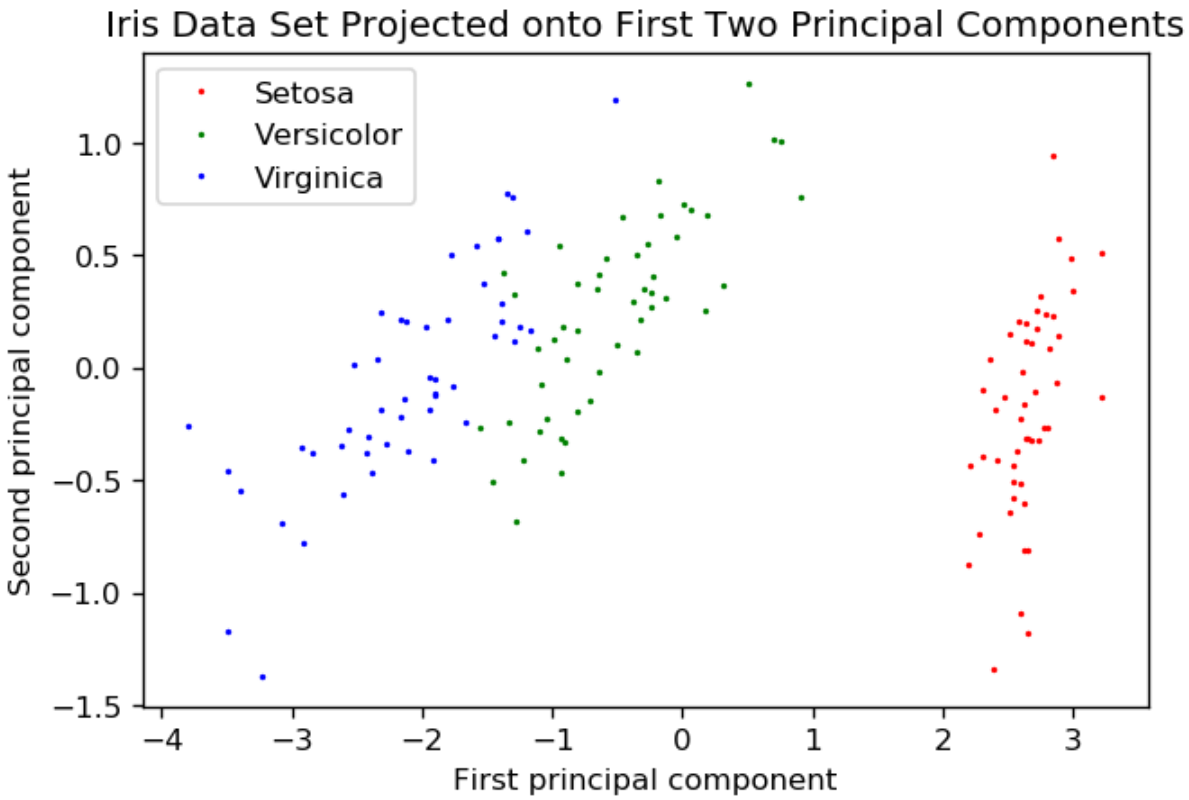
The first principal component $\mathbf{e}_0$ has the property that $X\mathbf{e}_0$ has the *highest possible variance* among all linear combination of the features in D. The second principal component $\mathbf{e}_1$ has the property that $X\mathbf{e}_1$ has the second highest possible variance among all linear combinations of the features in D that are orthogonal to $\mathbf{e}_0$, and so forth.

Using the normal projections $a_0, a_1, \ldots, a_{k-1}$ of $\mathbf{x} = \mathbf{d} - \mu$ onto the first k principal components $\mathbf{e}_0, \mathbf{e}_1, \ldots, \mathbf{e}_{k-1}$, we can obtain a low-dimensional view (and further usage) of any row vector $\mathbf{d}^T$ in D.

In this low-dimensional view of the data set, instead of p coordinates for every sample $\mathbf{d}$, we now have k coordinates: $a_0, a_1, \ldots, a_{k-1}$.

In [97]:
```
C = np.cov( D.T )                    # 4x4 covariance matrix.
print( "Covariance of the iris data set" )
print( C )

U, sigma, Vt = spla.svd( C )         # U and V are equivalent: they contain the eigenve
print( "\nShowing the first two principal components:" )
print( "Sepal Length:", U[0, :2] )
print( " Sepal Width:", U[1, :2] )
print( "Petal Length:", U[2, :2] )
print( " Petal Width:", U[3, :2] )

# Defining some indices.
setosaIdx = range( 0, 50 )           # The target classes appear in three consecutive r
versicolorIdx = range( 50, 100 )
virginicaIdx = range( 100, 150 )

# Compute the covariance matrix of the iris data set.
C = np.cov( D.T )

# Perform PCA on C.
U, sigma, Vt = spla.svd( C )
principalComponent0, principalComponent1 = U[:,0], U[:,1]

# Recall to center the data before projection.
m = np.mean( D, axis=0 )
X = D - m   # NumPy broadcasts the row vector into a matrix (no need to construct M!

# Project onto two first principal components
x = X @ principalComponent0
y = X @ principalComponent1

# Plotting projected data.
fig = plt.figure( dpi=120 )
ax = fig.add_subplot( 111 )
ax.set_title( "Iris Data Set Projected onto First Two Principal Components" )
ax.plot( x[setosaIdx], y[setosaIdx], "r.", markersize=2, label="Setosa" )
ax.plot( x[versicolorIdx], y[versicolorIdx], "g.", markersize=2, label="Versicolor" )
ax.plot( x[virginicaIdx], y[virginicaIdx], "b.", markersize=2, label="Virginica" )
ax.set_xlabel( "First principal component" )
ax.set_ylabel( "Second principal component" )
plt.legend()
plt.show()
```

```
Covariance of the iris data set
[[ 0.6857 -0.0424  1.2743  0.5163]
 [-0.0424  0.19   -0.3297 -0.1216]
 [ 1.2743 -0.3297  3.1163  1.2956]
 [ 0.5163 -0.1216  1.2956  0.581 ]]

Showing the first two principal components:
Sepal Length: [-0.3614 -0.6566]
 Sepal Width: [ 0.0845 -0.7302]
Petal Length: [-0.8567  0.1734]
 Petal Width: [-0.3583  0.0755]
```



Last, we prove by example that the singular values of the covariance matrix C are in fact the variances of the projected centered data. Suppose, again, that $X = D - M$ is the n-by-p matrix of centered data, and let U be the matrix of principal components of D. We then have that the first principal component $\mathbf{e}_0$ has the property that $X\mathbf{e}_0$ has the *highest possible variance* equivalent to the *largest singular value* σ_0. Then, $X\mathbf{e}_1$ has the *second highest possible variance* equivalent to the *second largest singular value* σ_1, and so forth.

In [98]:
```
print( "Singular values:", sigma )    # Singular values of the covariance matrix.
m = np.mean( D, axis=0 )
X = D - m
x = X @ principalComponent0 # Projection of the centered data onto the first principa
print( "Variance mthd 1: ", np.var( x, ddof=1 ) )   # One method to find the variance
print( "Variance mthd 2: ", 1 / (n - 1) * x.T @ x ) # Another method to find the vari
                                                    # Why is this second method true?
```

```
Singular values: [4.2282 0.2427 0.0782 0.0238]
Variance mthd 1:  4.228241706034864
Variance mthd 2:  4.228241706034864
```

Eigenfaces

We will create a data set F of gray-scale images. Each image will be a row in F, and their pixels will be placed in the columns.

The accompanying folder `faces/` has a data set of 178 black-and-white face images. Out of these 178 faces, there are a few of them (indicated in `face_descriptions.py`) that should be omitted. Each image is of size 64×64 pixels.

To build our face data set matrix F,

1. Reshape each i^{th} image into a row vector, $\mathbf{f}_i^T$, and
2. Stack all of our 1×64^2 vectors in F.

First, let's define some utility functions. These functions assume that images are gray-scale matrices of `uint8` values between 0 and 255, and vectors are double precision so that we can perform arithmetics on them.

By using these functions we can build the n-by-p matrix F with $n = 169$ faces stacked as row vectors with $p = 64^2$ columns. This way, the image vector $\mathbf{f}_i^T$ is at the i^{th} row of F:

$$F = \begin{matrix} \text{face image } 0 \\ \text{face image } 1 \\ \vdots \\ \text{face image } n-1 \end{matrix} \begin{pmatrix} f_{0,0} & f_{0,1} & \cdots & f_{0,p-1} \\ f_{1,0} & f_{1,1} & \cdots & f_{1,p-1} \\ \vdots & \vdots & \ddots & \vdots \\ f_{n-1,0} & f_{n-1,1} & \cdots & f_{n-1,p-1} \end{pmatrix} \in \mathbb{R}^{169 \times 64^2}.$$

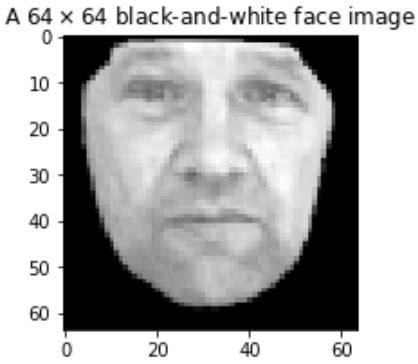
Let's next load all images in the `faces/` directory to build F.

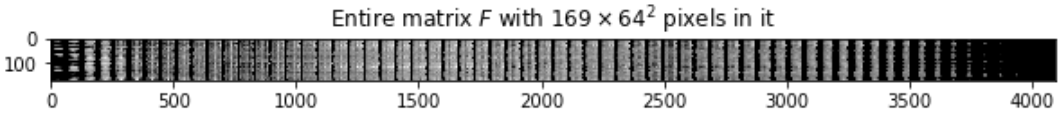
We can compute the average face as $\mathbf{m} = \frac{1}{n} \sum_i \mathbf{f}_i$.

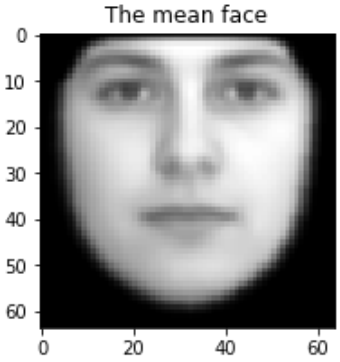
In [99]:
```
# Loading a black-and-white face image for its visualization.
Face = plt.imread( "faces/face000.bmp", format="bmp" )
print( "Face image has shape:", Face.shape )
print( "            and type:", Face.dtype )      # Face is a 64x64 matrix of gray sc
print( "       Minimum value:", np.min( Face ) )  # values thay may range between 0 a
print( "       Maximum value:", np.max( Face ) )

# Let's show the image.
plt.figure( figsize=(3,3) )
plt.title( r"A $64\times 64$ black-and-white face image" )
plt.imshow( Face, cmap="gray" )

# Some global variables.
HEIGHT = WIDTH = 64

################ Useful functions to go from images to vectors and back ##########
imageToVector = lambda Bitmap: Bitmap.reshape( -1 ).astype( np.float64 )
vectorToImage = lambda Vector: Vector.clip( 0, 255 ).reshape( HEIGHT, WIDTH ).astype(
renderVector = lambda Vector: plt.imshow( vectorToImage( Vector ), cmap="gray" )
scaleImageIntensities = lambda A: np.round( (A - np.min( A )) / (np.max( A ) - np.mir
##################################################################################

import face_descriptions   # Includes the file names we are interested in.

F = []                     # We append to F first as a list; then we make it into a r
for fileName in sorted( face_descriptions.face_features ):
    if fileName not in face_descriptions.image_to_omit:
        bitMap = plt.imread( "faces/" + fileName )
        F.append( imageToVector( bitMap ) )        # Flatenning the image.
F = np.array( F )          # Now F is a matrix.

print( "Shape of F is:", F.shape )

plt.figure( figsize=(10, 3) )
plt.title( r"Entire matrix $F$ with $169\times 64^2$ pixels in it" )
plt.imshow( F, cmap="gray" )

m = np.mean( F, axis=0 )   # Notice the axis=0 parameter!

# Plotting resulting mean face.
plt.figure( figsize=(3,3) )
plt.title( r"The mean face" )
renderVector( m )
```

```
Face image has shape: (64, 64)
            and type: uint8
       Minimum value: 0
       Maximum value: 182
Shape of F is: (169, 4096)
```

Out[99]:
```
<matplotlib.image.AxesImage at 0x7f9bd7614f50>
```







Let $\mathbf{m}^T$ be the mean row vector of matrix F as defined above. Suppose we stack $\mathbf{m}^T$, yet again, n times in a matrix M. Then, the covariance matrix of F is given by

$$C = \text{cov}(F) = \frac{1}{n-1}(F - M)^T(F - M)$$

If we define $X = (F - M)$, then

$$\text{cov}(X) = \frac{1}{n-1}X^T X = \frac{1}{n-1}(F - M)^T(F - M) = \text{cov}(F) = C,$$

and

$$C = USV^T = VSV^T,$$

since C is symmetric positive semidefinite and $U = V$. We call the columns in V the **principal components** of F, and we can use them to express any face $\mathbf{f}$. Suppose $\mathbf{x} = \mathbf{f} - \mathbf{m}$ is a vectorized/flattened face image with all pixel values centered around the corresponding means. We can express $\mathbf{x}$ as

$$\mathbf{x} = V\mathbf{c} = c_0\mathbf{e}_0 + c_1\mathbf{e}_1 + \cdots + c_{p-1}\mathbf{e}_{p-1},$$

where $\mathbf{c} = (c_0, c_1, \ldots, c_{p-1})^T$ is a vector of coefficients, and

$$V = \left(\begin{array}{c|c|c|c} & & & \\ \mathbf{e}_0 & \mathbf{e}_1 & \cdots & \mathbf{e}_{p-1} \\ & & & \end{array} \right)$$

is the p-by-p matrix of singular vectors of $\text{cov}(X)$.

To find the vector of coefficients/projections $\mathbf{c}$, we use

$$\mathbf{c} = V^T \mathbf{x},$$

and so $\mathbf{x} = V\mathbf{c} = V\left(V^T\mathbf{x}\right) = \mathbf{x}$. Furthermore, since $\mathbf{x} = \mathbf{f} - \mathbf{m}$, then

$$\mathbf{x} = (\mathbf{f} - \mathbf{m}) = V\mathbf{c},$$

which implies

$$\mathbf{f} = \mathbf{m} + V\mathbf{c}.$$

This is our *cookbook* to fabricate faces.

With PCA, we can perform data compression or low-rank approximation if we use only the first k singular vectors in V. If we do so with any face $\mathbf{f}$, then

$$\mathbf{f}^{(k)} = \mathbf{m} + V^{(k)}\mathbf{c} = \mathbf{m} + c_0\mathbf{e}_0 + c_1\mathbf{e}_1 + \cdots + c_{k-1}\mathbf{e}_{k-1}.$$

The singular vectors in V are called **eigenfaces**.

By modifying the coefficient of an eigenface (i.e., c_j for $\mathbf{e}_j$ above), we can alter the appearance of the face. Let's modify the coefficient for the fourth (i.e., $j = 3$) eigenface, which emphasizes the tip of the nose and the eyebrows. In general, the brighter the pixels in an eigenface, the more it emphasizes those pixels/features in a face.

By having our face *cookbook* (V = eigenfaces) at hand, we can approximate a face $\mathbf{f}$ with just k numbers!

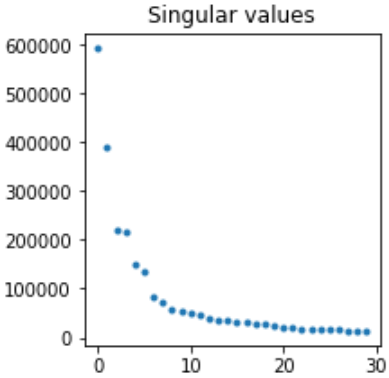
$$\mathbf{f}^{(k)} = \mathbf{m} + V^{(k)}\mathbf{c} = \mathbf{m} + c_0\mathbf{e}_0 + c_1\mathbf{e}_1 + \cdots + c_{k-1}\mathbf{e}_{k-1}.$$

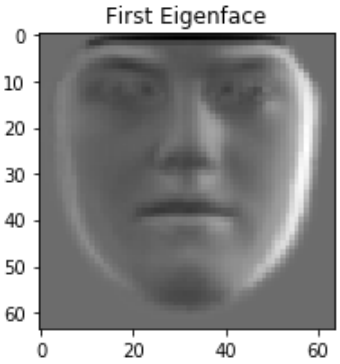
By knowing the mean face $\mathbf{m}$ and the eigenfaces in V, we can synthesize any face $\tilde{\mathbf{f}}$ by sampling a vector of coefficients $\tilde{\mathbf{c}}$ from some normal distribution:

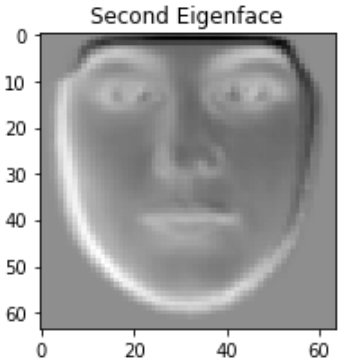
$$\tilde{\mathbf{f}}^{(k)} = \mathbf{m} + V^{(k)}\tilde{\mathbf{c}} = \mathbf{m} + \tilde{c}_0\mathbf{e}_0 + \tilde{c}_1\mathbf{e}_1 + \cdots + \tilde{c}_{k-1}\mathbf{e}_{k-1}.$$

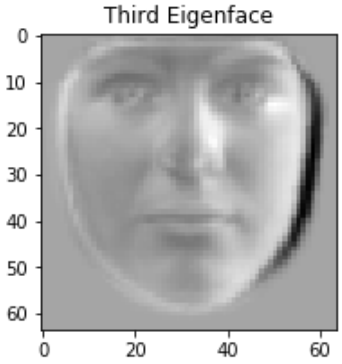
In [100]:
```
# We already know F and m.  Let's compute X: the centered matrix F around the column
X = F - m    # Notice that m will be broadcasted to fit the dimensions of X.

# Compute the covariance matrix of the the centered data.
C = np.cov( X.T )
print( "Shape of cov(X):", C.shape )

# Perform PCA on C using the SVD.
U, sigma, Vt = spla.svd( C )
Eigenfaces = Vt.T          # The Eigenfaces are the eigenvectors (U = V).

# Let's look at the first largest k eigen/singular values.
k = 30

# Plotting eigenfaces and the explained variance for first k singular values.
fig = plt.figure( figsize=(10,7) )
plt.subplot( 231 )
plt.title( "Singular values" )
plt.plot( sigma[:k], "." )
plt.subplot( 232 )
plt.title( "First Eigenface" )
renderVector( scaleImageIntensities( Eigenfaces[:,0] ) )
plt.subplot( 233 )
plt.title( "Second Eigenface" )
renderVector( scaleImageIntensities( Eigenfaces[:,1] ) )
plt.subplot( 234 )
plt.title( "Third Eigenface" )
renderVector( scaleImageIntensities( Eigenfaces[:,2] ) )
plt.subplot( 235 )
plt.title( "Fourth Eigenface" )
renderVector( scaleImageIntensities( Eigenfaces[:,3] ) )
plt.subplot( 236 )
plt.title( "Fifth Eigenface" )
renderVector( scaleImageIntensities( Eigenfaces[:,4] ) )
plt.show()

# At this point, we know the mean face m and the Eigenfaces V.
f = plt.imread( "faces/face000.bmp" )    # Let's modify the first face in the data se
f = imageToVector( f )
x = f - m
c = Vt @ x                               # Equivalently, Eigenfaces.T @ x: How many c

factors = [1, 3, 6, 9]
plt.figure( figsize=(10,4) )
subplotCode = 140
for i in range( len( factors ) ):
    factor = factors[i]
    plt.subplot( subplotCode + i + 1 )
    cNew = c.copy()
    cNew[3] *= factor
    renderVector( m + Eigenfaces @ cNew ) # Recall that Eigenfaces = V.
    plt.xlabel( "factor = {}".format( factor ) )
    if i == 1:
        plt.title( "A face image with 4th Eigenface coefficient multiplied by a const
```

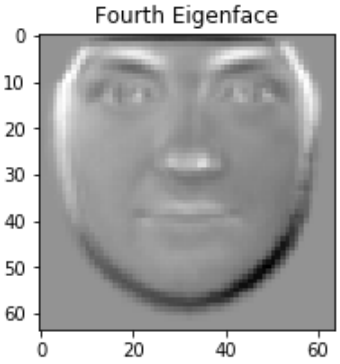
```
Shape of cov(X): (4096, 4096)
```

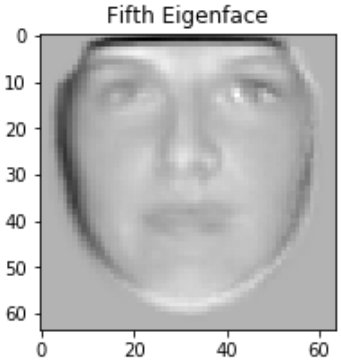

Singular values
600000
500000
400000
300000
200000
100000
0
0
10
20
30

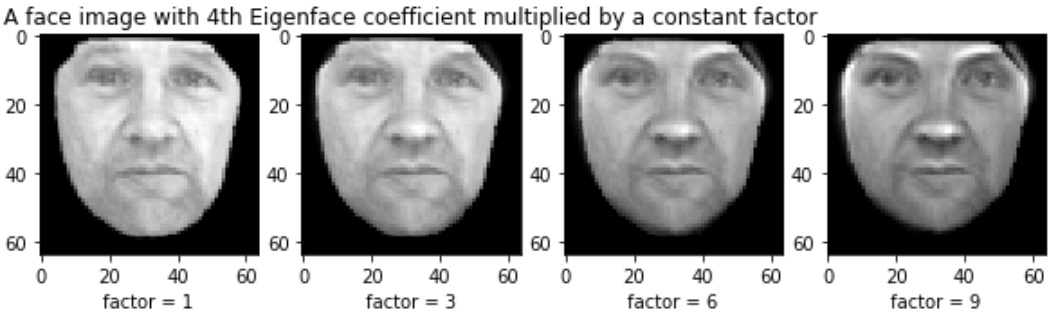

First Eigenface
0
10
20
30
40
50
60
0
20
40
60


Second Eigenface
0
10
20
30
40
50
60
0
20
40
60


Third Eigenface
0
10
20
30
40
50
60
0
20
40
60


Fourth Eigenface
0
10
20
30
40
50
60
0
20
40
60


Fifth Eigenface
0
10
20
30
40
50
60
0
20
40
60

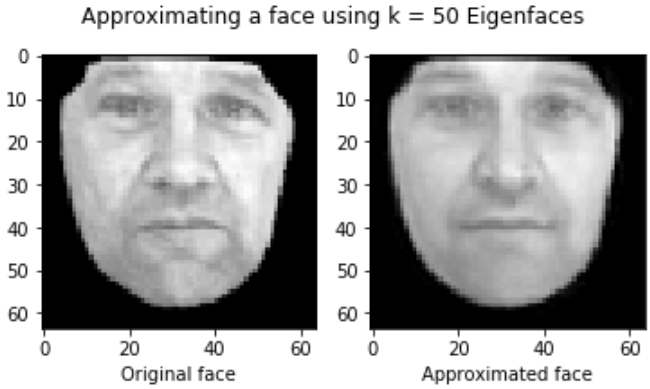

A face image with 4th Eigenface coefficient multiplied by a constant factor
0
20
40
60
0
20
40
60
factor = 1
factor = 3
factor = 6
factor = 9

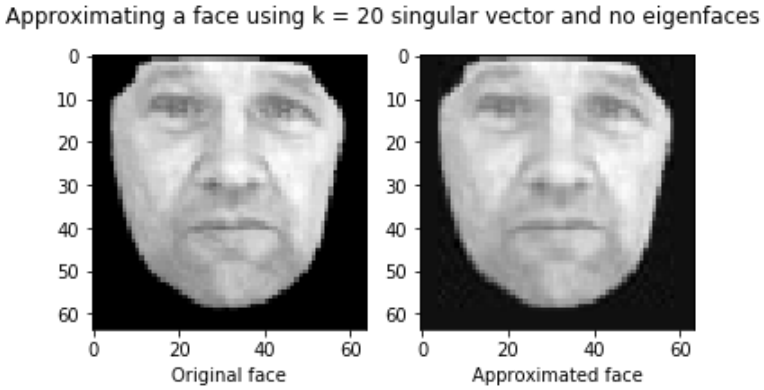
In [101]:
```
############## The kth approximation of a face from the cookbook ##############
k = 50                                      # You must use k <= 169 because min(row,col)
f = plt.imread( "faces/face000.bmp" )       # Let's take again the first face in the dat
f = imageToVector( f )
x = f - m
c = Eigenfaces[:,:k].T @ x

# Plotting the resulting kth approximation and the original image.
plt.figure( figsize=(6,3))
plt.subplot( 121 )
renderVector( f )
plt.xlabel( "Original face" )
plt.subplot( 122 )
renderVector( m + Eigenfaces[:,:k] @ c )
plt.xlabel( "Approximated face" )
plt.suptitle( "Approximating a face using k = {} Eigenfaces".format( k ) )

############## The kth approximation of a face *not* using Eigenfaces ##############
k = 20
f = plt.imread( "faces/face000.bmp" )       # Let's take again the first face in the dat
U2, sigma2, V2t = spla.svd( f )
fk = np.zeros( f.shape )
for i in range( k ):
    fk += sigma2[i] * np.outer( U2[:,i], V2t[i,:] )

plt.figure( figsize=(6,3))
plt.subplot( 121 )
plt.imshow( f, cmap="gray" )
plt.xlabel( "Original face" )
plt.subplot( 122 )
plt.imshow( fk, cmap="gray" )
plt.xlabel( "Approximated face" )
plt.suptitle( "Approximating a face using k = {} singular vector and no eigenfaces".f

############## Generating a random face from the cookbook ##############
k = 20                                      # Use only the top k Eigenfaces.
Coeffs = X @ Eigenfaces[:,:k]               # An nxk matrix of face coefficients.
mu = np.mean( Coeffs, axis=0 )              # Mean row vector of coefficients.
std = np.std( Coeffs, ddof=1, axis=0 )      # Standard deviations along each column of c

print( "Variance from function:" )          # Checking that the variances on coefficient
print( std ** 2 )                           # as the singular values!
print( "\nVariance from SVD:" )
print( sigma[:k] )

# To generate the new face, we must generate new coefficients based on the
# statistics from the population: the mean and std obtained above.
newFaceCoeffs = std * np.random.randn( k ) + mu  # This creates coefficients with a r
newFace = m + Eigenfaces[:,:k] @ newFaceCoeffs   # whose mean is mu and variance is s

# Plotting new random face.
plt.figure( figsize=(3,3) )
plt.title( "Random face" )
renderVector( newFace )
```

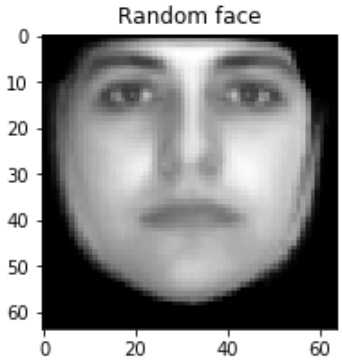
```
Variance from function:
[594316.7828 388739.2715 220110.72   215765.4029 148366.1161 136747.6608
  83176.0992  70937.4911  59014.928   55035.6332  49892.0923  45004.2084
  40254.1623  35670.4641  34394.6517  32401.9497  30830.3782  28070.3181
  26477.4945  24139.0365]

Variance from SVD:
[594316.7828 388739.2715 220110.72   215765.4029 148366.1161 136747.6608
```

```
  83176.0992  70937.4911  59014.928   55035.6332  49892.0923  45004.2084
  40254.1623  35670.4641  34394.6517  32401.9497  30830.3782  28070.3181
  26477.4945  24139.0365]
```

Out[101]: <matplotlib.image.AxesImage at 0x7f9ba6daee10>







Floating Point Representation

int64: integers from -2^{63} to $2^{63} - 1$ using 64 bits = 16 hexadecimal digits
float64: contains a **sign** , **mantissa** (fixed # of digits), **base** , and **exponent**
Normalized Number: one digit before decimal
IEEE Standard: 64 bits = 1 sign bit, 11 exponent bits (base 2), 52 fraction bits (for mantissa)
The exponent can be from 0 to 2047 (2^11 - 1):

- If exponent is between 1 and 2046, then number = sign (1.fraction) * 2^{exponent - 1023} (Note that the 1 before the decimal adds a free bit to the mantissa)
- If exponent = 0 and fraction = 0, then number = 0
- If exponent = 2047 and fraction = 0, then number = infinity
- If exponent = 2047 and fraction != 0, then number = NaN

Note: float64 has both +0 and -0, but they compare as equal
machine epsilon ε**:** the smallest `x` such that `1 + x > 1`. To find it, notice that `1` and `1 + x` differ only in the last bit of the mantissa (which has the last bit of `frac` set to `1`). This difference is equivalent to the real number 2^{-52}. As a floating-point number, ε has `exp = 3cb` and `frac` set to zeros.

Don't get confused by the meaning of "machine epsilon". It's **not** the smallest number you can represent in the floating-point system. To see why that's true, please check the last segment of this notebook.

Accuracy doesn't mean theat every time you do a computation, you get the right answer to 16 decimal digits.

Accuracy means that the floating-point system will do the best it can to give you an answer with 16 significant decimal figures.

The largest float64 has a mantissa of all ones, and the largest non-infinity `exp 7fe`. The second-largest float64 differs in the last bit of the mantissa.

The smallest normalized float64 has a mantissa of `1`, so its `frac` is all zero, and it has the smallest possible nonzero `exp`. The next larger normalized float64 has a `frac` of `1`.

The distance between those two numbers is smaller than any normalized float64. But it doesn't get rounded to zero -- it is a "denormal" float64. In fact, it's the smallest denormal float64, and thus the smallest positive float64 of all.

In [102]:
```
print(cs111.bits) #hex to binary

#Int64 to Hex
print( cs111.int64_to_hex( 2**63 - 1 ) ) # Largest positive int64
print( cs111.int64_to_hex( -2**63 ) ) # Most negative int64

#Floats
cs111.print_float64( 0 )

#Limits
print(-2 * np.inf)
print(0 * np.inf)

cs111.print_float64( np.inf )
cs111.print_float64( -np.inf )
cs111.print_float64( np.nan )
print(np.inf == np.inf)
print(np.nan == np.nan)
print(np.isnan( np.nan ))

#Machine Epsilon
epsilon = 2**(-52)   # Or 1/2**52.
print( "Machine epsilon:" )
cs111.print_float64( epsilon )
```

```
{'0': '0000', '1': '0001', '2': '0010', '3': '0011', '4': '0100', '5': '0101', '6':
'0110', '7': '0111', '8': '1000', '9': '1001', 'a': '1010', 'b': '1011', 'c': '110
0', 'd': '1101', 'e': '1110', 'f': '1111'}
7fffffffffffffff
8000000000000000
input     : 0
as float64: 0.0000000000000000e+00
as hex    : 0000000000000000
sign      : 0 means +
exponent  : 000 means zero or denormal

-inf
nan
input     : inf
as float64: inf
as hex    : 7ff0000000000000
sign      : 0 means +
exponent  : 7ff means inf or nan

input     : -inf
as float64: -inf
as hex    : fff0000000000000
sign      : 1 means -
exponent  : 7ff means inf or nan

input     : nan
as float64: nan
as hex    : 7ff8000000000000
sign      : 0 means +
exponent  : 7ff means inf or nan

True
False
True
Machine epsilon:
input     : 2.220446049250313e-16
as float64: 2.2204460492503131e-16
as hex    : 3cb0000000000000
```

```
sign      : 0 means +
exponent  : 3cb means 971 - 1023 = -52
mantissa  : 1.0000000000000000000000000000000000000000000000000000
```

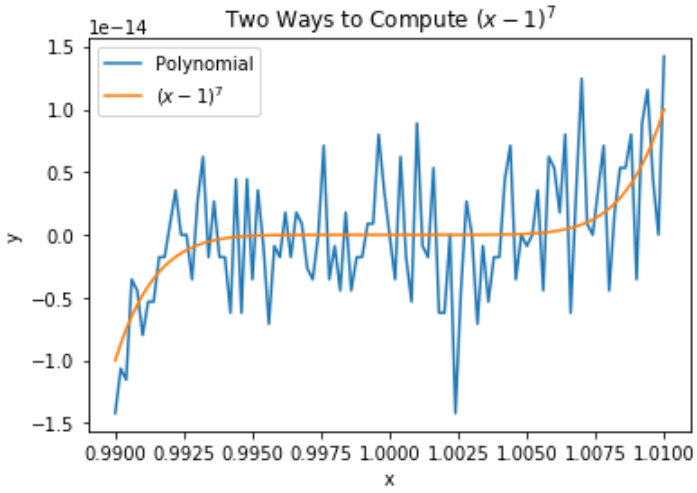
In [103]:
```
# Note this does not run infinitely, there's a limit (machine episilon):
x = 1.0
while 1 + x > 1:
    print( "x:", x, "    1 + x:", 1 + x )
    x = x / 2
print( "x:", x, "    1 + x:", 1 + x )
```

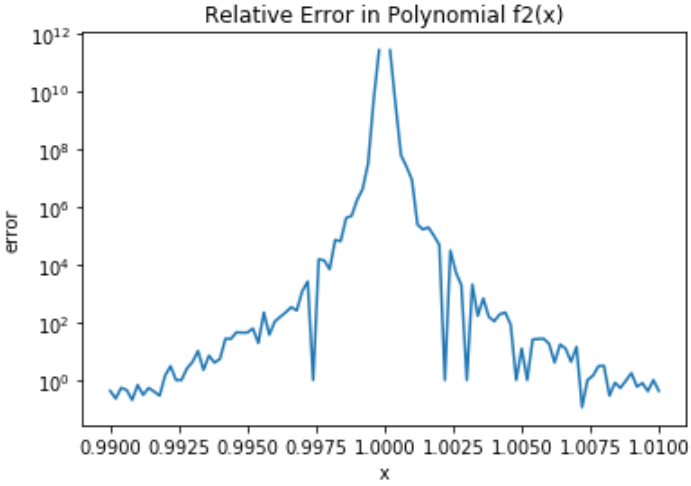
```
x: 1.0     1 + x: 2.0
x: 0.5     1 + x: 1.5
x: 0.25     1 + x: 1.25
x: 0.125     1 + x: 1.125
x: 0.0625     1 + x: 1.0625
x: 0.03125     1 + x: 1.03125
x: 0.015625     1 + x: 1.015625
x: 0.0078125     1 + x: 1.0078125
x: 0.00390625     1 + x: 1.00390625
x: 0.001953125     1 + x: 1.001953125
x: 0.0009765625     1 + x: 1.0009765625
x: 0.00048828125     1 + x: 1.00048828125
x: 0.000244140625     1 + x: 1.000244140625
x: 0.0001220703125     1 + x: 1.0001220703125
x: 6.103515625e-05     1 + x: 1.00006103515625
x: 3.0517578125e-05     1 + x: 1.000030517578125
x: 1.52587890625e-05     1 + x: 1.0000152587890625
x: 7.62939453125e-06     1 + x: 1.0000076293945312
x: 3.814697265625e-06     1 + x: 1.0000038146972656
x: 1.9073486328125e-06     1 + x: 1.0000019073486328
x: 9.5367431640625e-07     1 + x: 1.0000009536743164
x: 4.76837158203125e-07     1 + x: 1.0000004768371582
x: 2.384185791015625e-07     1 + x: 1.000000238418579
x: 1.1920928955078125e-07     1 + x: 1.0000001192092896
x: 5.960464477539063e-08     1 + x: 1.0000000596046448
x: 2.9802322387695312e-08     1 + x: 1.0000000298023224
x: 1.4901161193847656e-08     1 + x: 1.0000000149011612
x: 7.450580596923828e-09     1 + x: 1.0000000074505806
x: 3.725290298461914e-09     1 + x: 1.0000000037252903
x: 1.862645149230957e-09     1 + x: 1.0000000018626451
x: 9.313225746154785e-10     1 + x: 1.0000000009313226
x: 4.656612873077393e-10     1 + x: 1.0000000004656613
x: 2.3283064365386963e-10     1 + x: 1.0000000002328306
x: 1.1641532182693481e-10     1 + x: 1.0000000001164153
x: 5.820766091346741e-11     1 + x: 1.0000000000582077
x: 2.9103830456733704e-11     1 + x: 1.0000000000291038
x: 1.4551915228366852e-11     1 + x: 1.000000000014552
x: 7.275957614183426e-12     1 + x: 1.000000000007276
x: 3.637978807091713e-12     1 + x: 1.000000000003638
x: 1.8189894035458565e-12     1 + x: 1.000000000001819
x: 9.094947017729282e-13     1 + x: 1.0000000000009095
x: 4.547473508864641e-13     1 + x: 1.0000000000004547
x: 2.2737367544323206e-13     1 + x: 1.0000000000002274
x: 1.1368683772161603e-13     1 + x: 1.0000000000001137
x: 5.684341886080802e-14     1 + x: 1.0000000000000568
x: 2.842170943040401e-14     1 + x: 1.0000000000000284
x: 1.4210854715202004e-14     1 + x: 1.0000000000000142
x: 7.105427357601002e-15     1 + x: 1.000000000000007
x: 3.552713678800501e-15     1 + x: 1.0000000000000036
x: 1.7763568394002505e-15     1 + x: 1.0000000000000018
x: 8.881784197001252e-16     1 + x: 1.0000000000000009
x: 4.440892098500626e-16     1 + x: 1.0000000000000004
x: 2.220446049250313e-16     1 + x: 1.0000000000000002
x: 1.1102230246251565e-16     1 + x: 1.0
```

In [104]:
```
# Catastrophic Cancellation: Computing (x-1)^7

# A simple function of a real number x.
def f1( x ):
    return ( x - 1 )**7

# Same function as f1, but multiplied out and written as a polynomial in x.
def f2(x):
    return x**7 - 7*x**6 + 21*x**5 - 35*x**4 + 35*x**3 - 21*x**2 + 7*x - 1

xvals = np.linspace( 0.99, 1.01, 101 )

y1 = []
y2 = []
for x in xvals:
    y1.append( f1( x ) )
    y2.append( f2( x ) )
y1 = np.array( y1 )
y2 = np.array( y2 )

plt.figure()
plt.plot( xvals, y2, label = "Polynomial" )
plt.plot( xvals, y1, label = r"$(x-1)^7$" )
plt.xlabel( "x" )
plt.ylabel( "y" )
plt.title( r"Two Ways to Compute $(x-1)^7$" )
plt.legend()

plt.figure()
plt.semilogy( xvals, np.abs( (y1 - y2) / y1 ) )
plt.xlabel( "x" )
plt.ylabel( "error" )
plt.title( "Relative Error in Polynomial f2(x)" )
```

```
/Library/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:30: RuntimeWar
ning: invalid value encountered in true_divide
```

Out[104]:
```
Text(0.5, 1.0, 'Relative Error in Polynomial f2(x)')
```




Relative Error in Polynomial f2(x)
error
x
10^{12}
10^{10}
10^{8}
10^{6}
10^{4}
10^{2}
10^{0}
0.9900 0.9925 0.9950 0.9975 1.0000 1.0025 1.0050 1.0075 1.0100

In [105]:
```
#Largest Floats
fMax = ((1 << 53) - 1) / 2**52 * 2**1023
print( "Largest float64:" )
cs111.print_float64( fMax )
print()

fNextMax = ((1 << 53) - 2) / 2**52 * 2**1023
print( "Second largest float64:" )
cs111.print_float64( fNextMax )
print()

print( "Difference:" )
cs111.print_float64( fMax - fNextMax )

#Smallest Floats
fMin = 1 * 2**(-1022)
print( "Smallest normalized float64:" )
cs111.print_float64( fMin )
print()

fNextMin = (1 + 1/2**52) * 2**(-1022)
print( "Second smallest normalized float64:" )
cs111.print_float64( fNextMin )
print()

print( "Difference (a denormal float64):" )
cs111.print_float64( fNextMin - fMin )
```

```
Largest float64:
input     : 1.7976931348623157e+308
as float64: 1.7976931348623157e+308
as hex    : 7fefffffffffffff
sign      : 0 means +
exponent  : 7fe means 2046 - 1023 = 1023
mantissa  : 1.1111111111111111111111111111111111111111111111111111


Second largest float64:
input     : 1.7976931348623155e+308
as float64: 1.7976931348623155e+308
as hex    : 7feffffffffffffe
sign      : 0 means +
exponent  : 7fe means 2046 - 1023 = 1023
mantissa  : 1.1111111111111111111111111111111111111111111111111110


Difference:
input     : 1.99584030953472e+292
as float64: 1.9958403095347198e+292
as hex    : 7ca0000000000000
sign      : 0 means +
exponent  : 7ca means 1994 - 1023 = 971
mantissa  : 1.0000000000000000000000000000000000000000000000000000

Smallest normalized float64:
input     : 2.2250738585072014e-308
as float64: 2.2250738585072014e-308
as hex    : 0010000000000000
sign      : 0 means +
exponent  : 001 means 1 - 1023 = -1022
mantissa  : 1.0000000000000000000000000000000000000000000000000000
```

```
Second smallest normalized float64:
input     : 2.225073858507202e-308
as float64: 2.2250738585072019e-308
as hex    : 0010000000000001
sign      : 0 means +
exponent  : 001 means 1 - 1023 = -1022
mantissa  : 1.0000000000000000000000000000000000000000000000000000
```

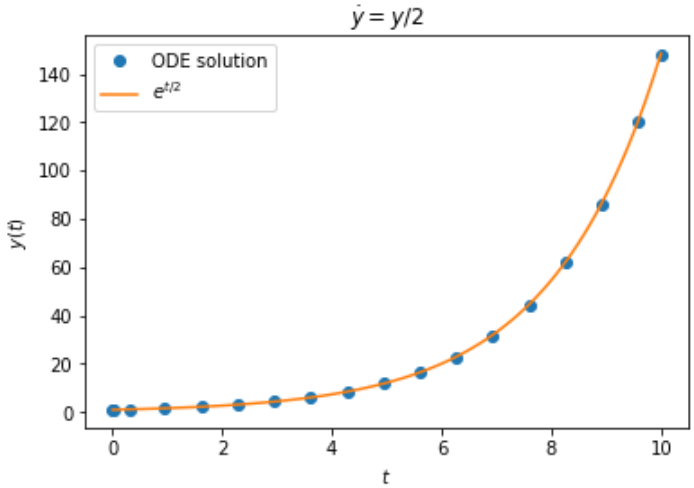
```
Difference (a denormal float64):
input     : 5e-324
as float64: 4.9406564584124654e-324
as hex    : 0000000000000001
sign      : 0 means +
exponent  : 000 means zero or denormal
```

Ordinary Differential Equations

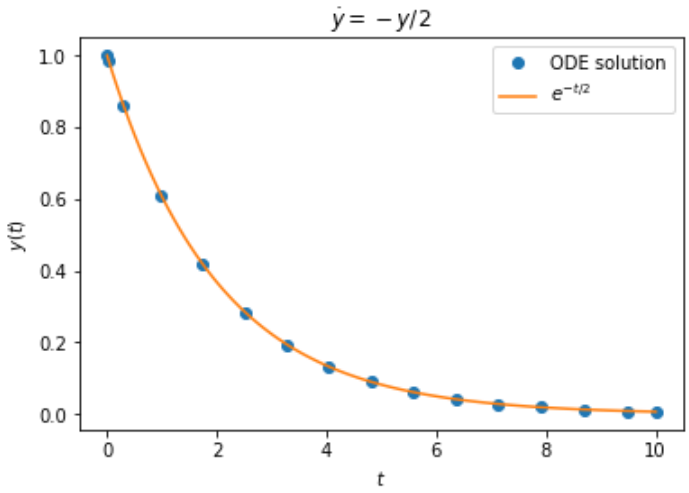
In [106]:
```
integrate.solve_ivp?
```

In [107]:
```
# Problem: dy/dt = (1/2) * y(t), superimposing exact solution.
# y(0) = 1
def f( t, y ):
    """
    Function to be integrated to solve an ODE or a system of ODEs.
    Input:
      t is a scalar time.
      y is a vector of variables.
    Output:
      yDot is the vector dy/dt.
    """
    yDot = y/2
    return yDot

tSpan = (0, 10)
yInit = [1]

sol = integrate.solve_ivp( fun=f, t_span=tSpan, y0=yInit, method="RK23" )

plt.plot( sol.t, sol.y[0], "o", label="ODE solution" )
tt = np.linspace( 0, 10, 100 )
plt.plot( tt, np.exp( tt / 2 ), label=r"$e^{t/2}$" )
plt.legend()
plt.xlabel( r"$t$" )
plt.ylabel( r"$y(t)$" )
plt.title( r"$\dot{y} = y/2$" )
print(sol)
```

```
  message: 'The solver successfully reached the end of the integration interval.'
     nfev: 53
     njev: 0
      nlu: 0
      sol: None
   status: 0
  success: True
        t: array([ 0.    ,  0.0272,  0.2987,  0.9549,  1.6188,  2.283 ,  2.9471,
        3.6112,  4.2752,  4.9393,  5.6034,  6.2674,  6.9315,  7.5955,
        8.2595,  8.9236,  9.5876, 10.    ])
 t_events: None
        y: array([[  1.    ,   1.0137,   1.1611,   1.6113,   2.2449,   3.1278,
          4.3579,   6.0717,   8.4595,  11.7863,  16.4212,  22.8788,
         31.8757,  44.4105,  61.8744,  86.2057, 120.1051, 147.5984]])
 y_events: None
```

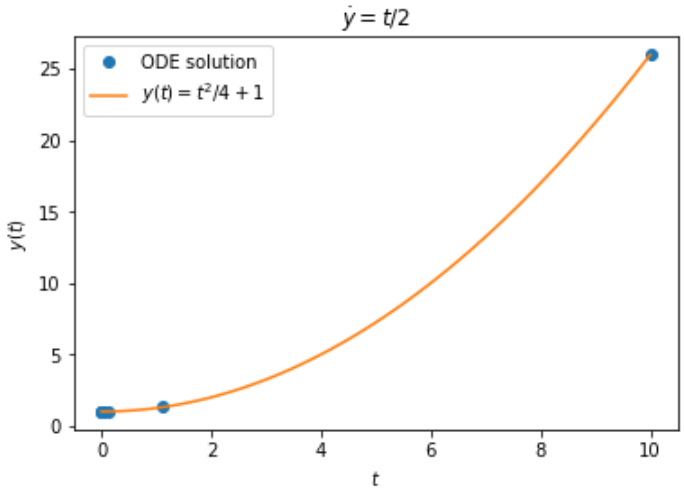


In [108]:
```
# Problem: dy/dt = -(1/2) * y(t)
# y(0) = 1
def f( t, y ):
    """
    Function to be integrated to solve an ODE or a system of ODEs.
    Input:
      t is a scalar time.
      y is a vector of variables.
    Output:
      yDot is the vector dy/dt.
    """
    yDot = -y/2
    return yDot

tSpan = (0, 10)
yInit = [1]

sol = integrate.solve_ivp( fun=f, t_span=tSpan, y0=yInit, method="RK23" )

plt.plot( sol.t, sol.y[0], "o", label="ODE solution" )
tt = np.linspace( 0, 10, 100 )
plt.plot( tt, np.exp( -tt / 2 ), label=r"$e^{-t/2}$" )
plt.legend()
plt.xlabel( r"$t$" )
plt.ylabel( r"$y(t)$" )
plt.title( r"$\dot{y} = -y/2$" )
print()
```

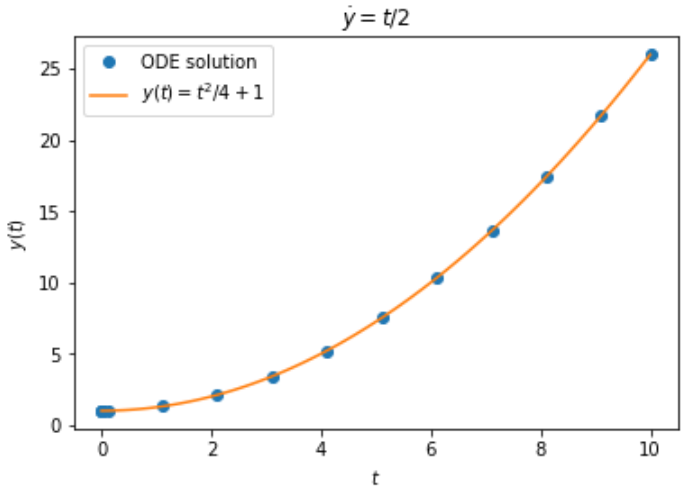


In [109]:
```
# Problem: dy/dt = t/2
# y(0) = 1
def f( t, y ):
    """
    Function to be integrated to solve an ODE or a system of ODEs.
    Input:
      t is a scalar time.
      y is a vector of variables.
    Output:
      yDot is the vector dy/dt.
    """
    yDot = t / 2
    return yDot

tSpan = (0, 10)
yInit = [1]

sol = integrate.solve_ivp( fun=f, t_span=tSpan, y0=yInit, method="RK23" )

plt.plot( sol.t, sol.y[0], "o", label="ODE solution" )
tt = np.linspace( 0, 10, 100 )
plt.plot( tt, tt ** 2 / 4 + 1, label=r"$y(t) = t^2/4 + 1$" )
plt.legend()
plt.xlabel( r"$t$" )
plt.ylabel( r"$y(t)$" )
plt.title( r"$\dot{y} = t/2$" )
print(sol.t)
```

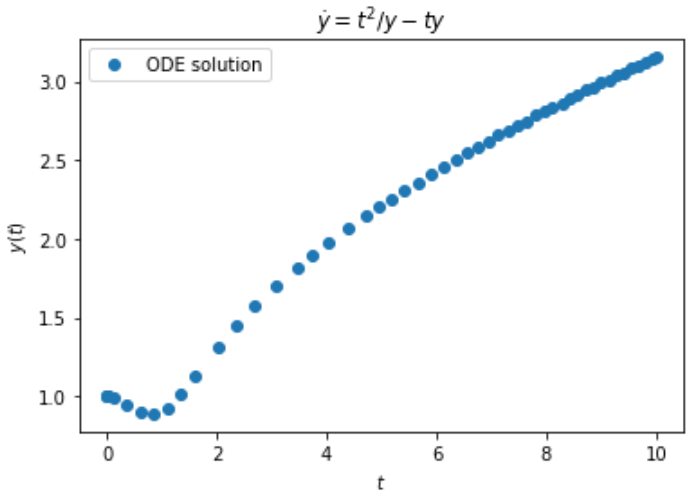
```
[0.0000e+00 1.0000e-04 1.1000e-03 1.1100e-02 1.1110e-01 1.1111e+00
 1.0000e+01]
```



In [110]:
```
# Problem: dy/dt = t/2, with max_step (prevent big steps)
# y(0) = 1
def f( t, y ):
    """
    Function to be integrated to solve an ODE or a system of ODEs.
    Input:
      t is a scalar time.
      y is a vector of variables.
    Output:
      yDot is the vector dy/dt.
    """
    yDot = t / 2
    return yDot

tSpan = (0, 10)
yInit = [1]

sol = integrate.solve_ivp( fun=f, t_span=tSpan, y0=yInit, method="RK23", max_step=1 )

plt.plot( sol.t, sol.y[0], "o", label="ODE solution" )
tt = np.linspace( 0, 10, 100 )
plt.plot( tt, tt ** 2 / 4 + 1, label=r"$y(t) = t^2/4 + 1$" )
plt.legend()
plt.xlabel( r"$t$" )
plt.ylabel( r"$y(t)$" )
plt.title( r"$\dot{y} = t/2$" )
print()
```

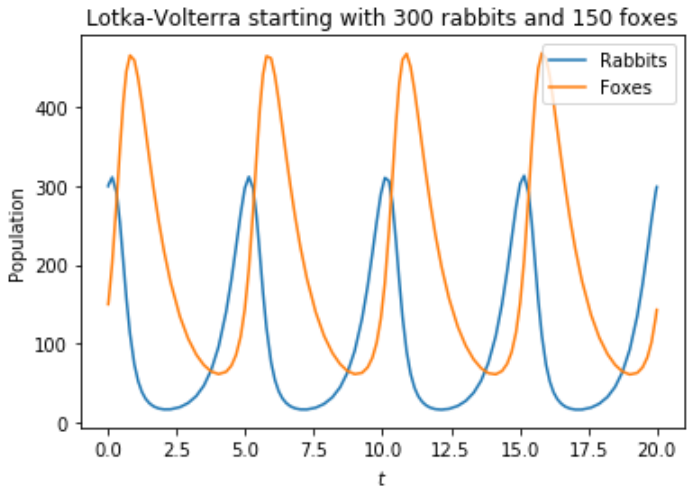


In [111]:
```
# An ODE with a hard-to-find analytical solution.
def f(t, y):
    """
    Function to be integrated to solve an ODE or a system of ODEs.
    Input:
      t is a scalar time.
      y is a vector of variables.
    Output:
      yDot is the vector dy/dt
    """
    yDot = t**2 / y - t * y
    return yDot

tSpan = (0, 10)
yInit = [1]

sol = integrate.solve_ivp( fun=f, t_span=tSpan, y0=yInit, method="RK23" )

plt.plot( sol.t, sol.y[0], "o", label="ODE solution" )
plt.legend()
plt.xlabel( r"$t$" )
plt.ylabel( r"$y(t)$" )
plt.title( r"$\dot{y} = t^2 / y - ty$" )
print()
```

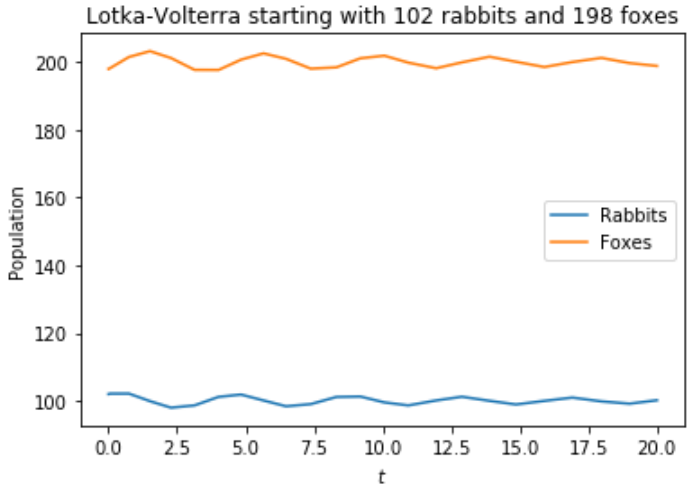


Systems of ODEs

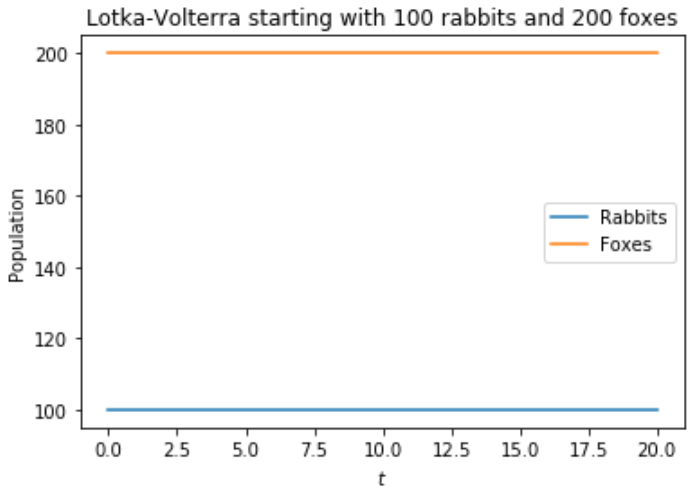
In [112]:
```
# Demo of Lotka-Volterra equations.
def lotka( t, y ):
    """
    Function to be integrated to solve the Lotka-Volterra equations.
    Input:
      t is time (scalar).
      y is [rabbits(t), foxes(t)].
    Output:
      yDot is the vector dy/dt.
    """
    alpha = .01
    yDot = [2*y[0] - alpha*y[0]*y[1],
             -y[1] + alpha*y[0]*y[1]]
    return yDot

tSpan = (0, 20)
rabbits_0 = 300
foxes_0 = 150
yInit = [rabbits_0, foxes_0]

sol = integrate.solve_ivp( fun=lotka, t_span=tSpan, y0=yInit, method="RK23" )

plt.plot( sol.t, sol.y[0], label="Rabbits" )
plt.plot( sol.t, sol.y[1], label="Foxes" )
plt.legend()
plt.xlabel( r"$t$" )
plt.ylabel( "Population" )
plt.title( "Lotka-Volterra starting with {} rabbits and {} foxes".format( rabbits_0,
print()
```



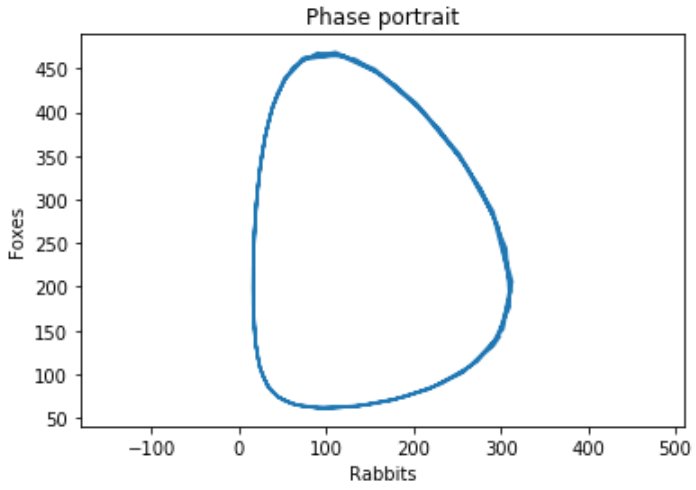
In [113]:
```
# Demo of Lotka-Volterra equations.
def lotka( t, y ):
    """
    Function to be integrated to solve the Lotka-Volterra equations.
    Input:
      t is time (scalar).
      y is [rabbits(t), foxes(t)].
    Output:
      yDot is the vector dy/dt.
    """
    alpha = .01
    yDot = [2*y[0] - alpha*y[0]*y[1],
             -y[1] + alpha*y[0]*y[1]]
    return yDot

tSpan = (0, 20)
rabbits_0 = 102
foxes_0 = 198
yInit = [rabbits_0, foxes_0]

sol = integrate.solve_ivp( fun=lotka, t_span=tSpan, y0=yInit, method="RK23" )

plt.plot( sol.t, sol.y[0], label="Rabbits" )
plt.plot( sol.t, sol.y[1], label="Foxes" )
plt.legend()
plt.xlabel( r"$t$" )
plt.ylabel( "Population" )
plt.title( "Lotka-Volterra starting with {} rabbits and {} foxes".format( rabbits_0,
print()
```



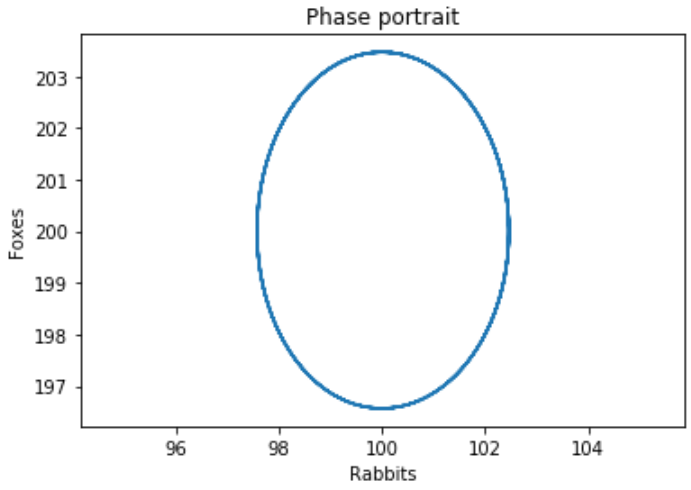
In [114]:
```
# Demo of Lotka-Volterra equations.
def lotka( t, y ):
    """
    Function to be integrated to solve the Lotka-Volterra equations.
    Input:
      t is time (scalar).
      y is [rabbits(t), foxes(t)].
    Output:
      yDot is the vector dy/dt.
    """
    alpha = .01
    yDot = [2*y[0] - alpha*y[0]*y[1],
             -y[1] + alpha*y[0]*y[1]]
    return yDot

tSpan = (0, 20)
rabbits_0 = 100
foxes_0 = 200
yInit = [rabbits_0, foxes_0]

sol = integrate.solve_ivp( fun=lotka, t_span=tSpan, y0=yInit, method="RK23" )

plt.plot( sol.t, sol.y[0], label="Rabbits" )
plt.plot( sol.t, sol.y[1], label="Foxes" )
plt.legend()
plt.xlabel( r"$t$" )
plt.ylabel( "Population" )
plt.title( "Lotka-Volterra starting with {} rabbits and {} foxes".format( rabbits_0,
print()
```



In [115]:
```
# Demo of Lotka-Volterra equations in a phase portrait.
def lotka( t, y ):
    """
    Function to be integrated to solve the Lotka-Volterra equations.
    Input:
      t is time (scalar).
      y is [rabbits(t), foxes(t)].
    Output:
      yDot is the vector dy/dt.
    """
    alpha = .01
    yDot = [2*y[0] - alpha*y[0]*y[1],
              -y[1] + alpha*y[0]*y[1]]
    return yDot

tSpan = (0, 20)
rabbits_0 = 300
foxes_0 = 150
yInit = [rabbits_0, foxes_0]

sol = integrate.solve_ivp( fun=lotka, t_span=tSpan, y0=yInit, method="RK23" )

plt.plot( sol.y[0], sol.y[1] )
plt.axis( "equal" )
plt.xlabel( "Rabbits" )
plt.ylabel( "Foxes" )
plt.title( "Phase portrait" )
print()
```

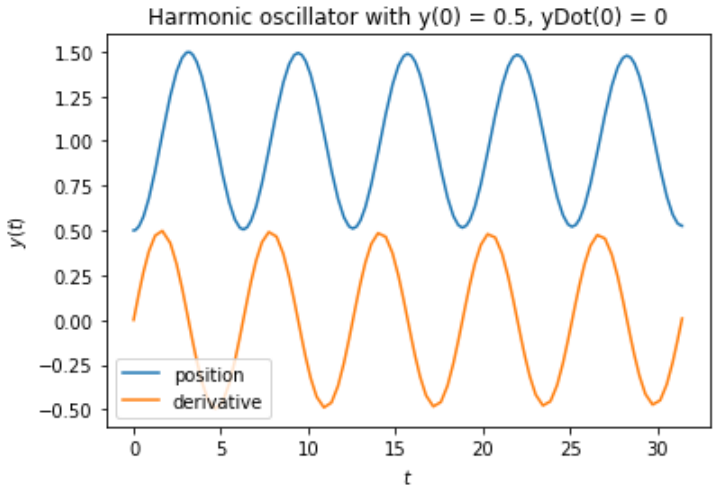


In [116]:
```
# Demo of Lotka-Volterra equations in a phase portrait.
def lotka( t, y ):
    """
    Function to be integrated to solve the Lotka-Volterra equations.
    Input:
      t is time (scalar).
      y is [rabbits(t), foxes(t)].
    Output:
      yDot is the vector dy/dt.
    """
    alpha = .01
    yDot = [2*y[0] - alpha*y[0]*y[1],
              -y[1] + alpha*y[0]*y[1]]
    return yDot

tSpan = (0, 20)
rabbits_0 = 102
foxes_0 = 198
yInit = [rabbits_0, foxes_0]

sol = integrate.solve_ivp( max_step=0.01, fun=lotka, t_span=tSpan, y0=yInit, method='

plt.plot( sol.y[0], sol.y[1] )
plt.axis( "equal" )
plt.xlabel( "Rabbits" )
plt.ylabel( "Foxes" )
plt.title( "Phase portrait" )
print()
```

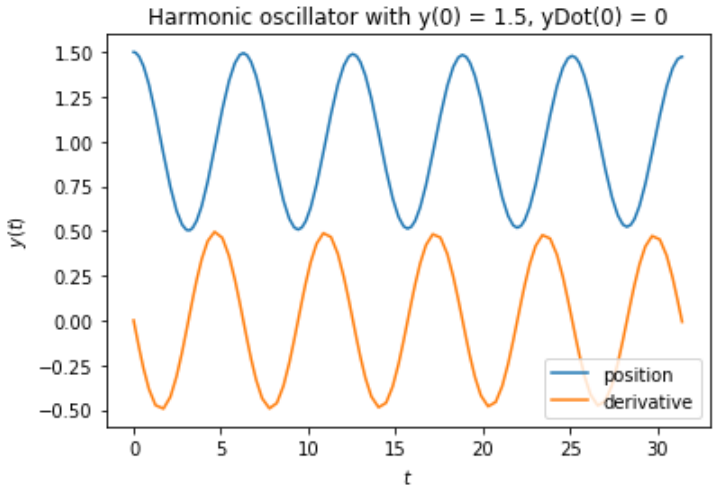


Higher Order ODEs

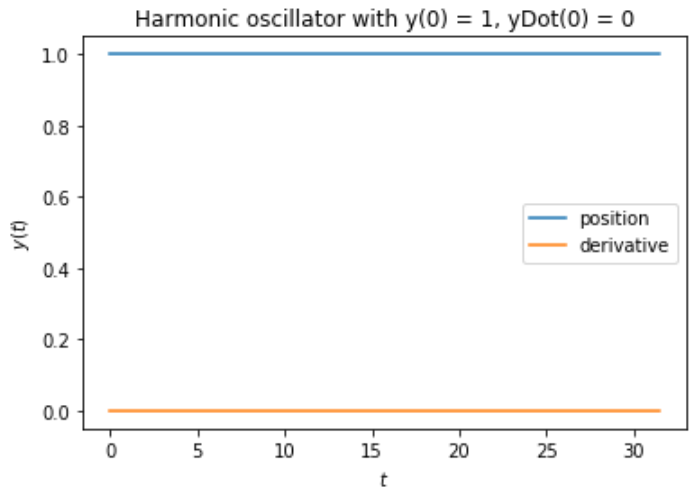
In [117]:
```
def weightf( t, y ):
    """
    Function to be integrated to solve a second-order ODE for a harmonic oscillator.
    Input:
      t is a scalar time,
      y is a vector of variables,
        in this case y = [position, 1st derivative]
    Output:
      yDot is the vector dy/dt,
        in this case yDot = [1st derivative, 2nd derivative]

    The second-order ODE being integrated is d^2 y / dt^2 = 1 - y.
    """
    yDot = [y[1] , 1 - y[0]]
    return yDot

tSpan = (0, 10 * np.pi)
yInit = [1/2, 0]
# Also try:
# yInit = [1.5, 0]
# yInit = [1, 0]
# yInit = [1, -1/4]

sol = integrate.solve_ivp( fun=weightf, t_span=tSpan, y0=yInit, method="RK23" )

plt.plot( sol.t, sol.y[0], label="position" )
plt.plot( sol.t, sol.y[1], label="derivative" )
plt.legend()
plt.xlabel( r"$t$" )
plt.ylabel( r"$y(t)$" )
plt.title( "Harmonic oscillator with y(0) = {}, yDot(0) = {}".format( yInit[0], yInit
print()
```



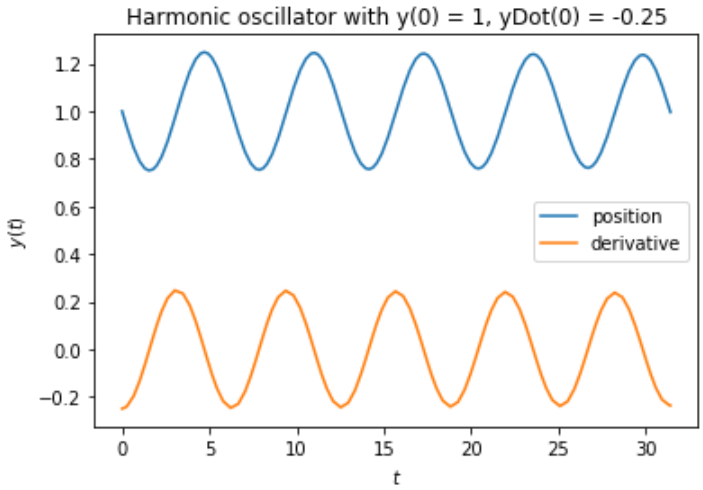
In [118]:
```
# yInit = [1/2, 0]
# Also try:
yInit = [1.5, 0]
# yInit = [1, 0]
# yInit = [1, -1/4]

sol = integrate.solve_ivp( fun=weightf, t_span=tSpan, y0=yInit, method="RK23" )

plt.plot( sol.t, sol.y[0], label="position" )
plt.plot( sol.t, sol.y[1], label="derivative" )
plt.legend()
plt.xlabel( r"$t$" )
plt.ylabel( r"$y(t)$" )
plt.title( "Harmonic oscillator with y(0) = {}, yDot(0) = {}".format( yInit[0], yInit
print()
```



In [119]:
```
tSpan = (0, 10 * np.pi)
# yInit = [1/2, 0]
# Also try:
# yInit = [1.5, 0]
yInit = [1, 0]
# yInit = [1, -1/4]

sol = integrate.solve_ivp( fun=weightf, t_span=tSpan, y0=yInit, method="RK23" )

plt.plot( sol.t, sol.y[0], label="position" )
plt.plot( sol.t, sol.y[1], label="derivative" )
plt.legend()
plt.xlabel( r"$t$" )
plt.ylabel( r"$y(t)$" )
plt.title( "Harmonic oscillator with y(0) = {}, yDot(0) = {}".format( yInit[0], yInit
print()
```



In [120]:
```
tSpan = (0, 10 * np.pi)
# yInit = [1/2, 0]
# Also try:
# yInit = [1.5, 0]
# yInit = [1, 0]
yInit = [1, -1/4]

sol = integrate.solve_ivp( fun=weightf, t_span=tSpan, y0=yInit, method="RK23" )

plt.plot( sol.t, sol.y[0], label="position" )
plt.plot( sol.t, sol.y[1], label="derivative" )
plt.legend()
plt.xlabel( r"$t$" )
plt.ylabel( r"$y(t)$" )
plt.title( "Harmonic oscillator with y(0) = {}, yDot(0) = {}".format( yInit[0], yInit
print()
```

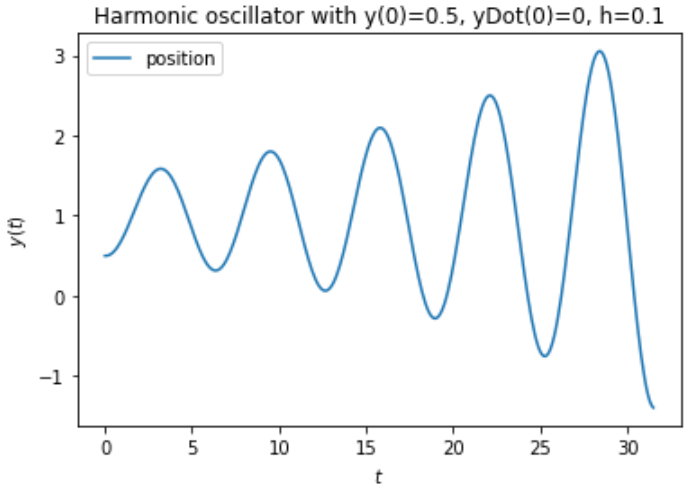


Euler's Method

In [121]:
```
def ode1( fun, t_span, y0, h ):
    """
    Forward Euler algorithm: demo version.
    ode1( fun, t_span, y0, h ) uses fixed step size h.
    This is just demo code, don't use it for real!
    """
    # First make the inputs into numpy arrays.
    t0 = np.array( t_span[0] ).reshape( 1 )
    tFinal = np.array(t_span[1]).reshape( 1 )
    y0 = np.array( y0 ).reshape( len( y0 ), 1 )   # A true column vector.

    # Initialize the list of solution points.
    sol_t = t0
    sol_y = y0

    step = 0
    t = t0
    y = y0
    while t < tFinal:
        s1 = np.array( fun( t, y ) )   # Slope.
        y = y + h * s1
        t = t + h
        sol_t = np.concatenate( (sol_t, t) )
        sol_y = np.concatenate( (sol_y, y), axis=1 )
        step += 1

    print( "ode1 took", step, "steps" )
    return sol_t, sol_y
```

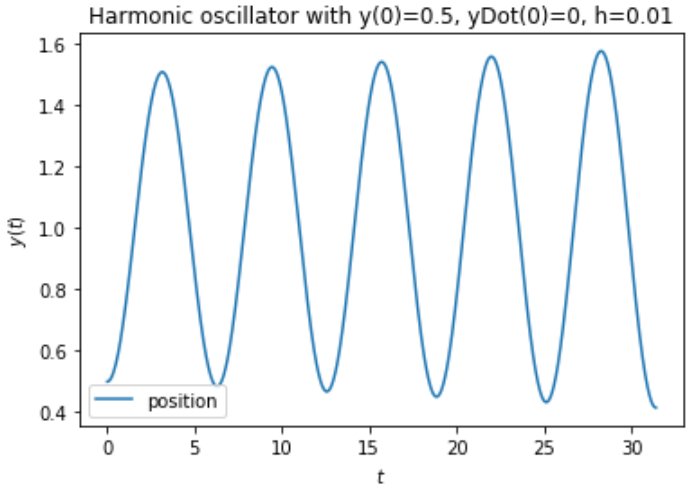
In [122]:
```
# Demo of ode1 with harmonic oscillator.
def weightf( t, y ):
    """
    Function to be integrated to solve a second-order ODE for a harmonic oscillator.
    Input:
      t is a scalar time,
      y is a vector of variables,
        in this case y = [position, 1st derivative]
    Output:
      yDot is the vector dy/dt,
        in this case ydot = [1st derivative, 2nd derivative]

    The second-order ODE being integrated is d^2 y / dt^2 = 1 - y.
    """
    yDot = [y[1] , 1 - y[0]]
    return yDot

tSpan = (0, 10 * np.pi)
yInit = [1/2, 0]
stepSize = .1   # Try .1, .01, .001.

sol_t, sol_y = ode1( fun=weightf, t_span=tSpan, y0=yInit, h=stepSize )

plt.plot( sol_t, sol_y[0], label="position" )
plt.legend()
plt.xlabel( r"$t$" )
plt.ylabel( r"$y(t)$" )
plt.title( "Harmonic oscillator with y(0)={}, yDot(0)={}, h={}".format( yInit[0], yIr
print()
```

```
ode1 took 315 steps
```



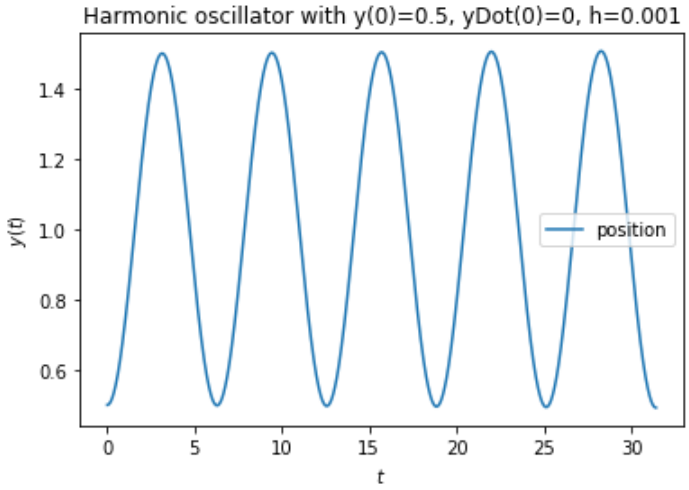
In [123]:
```
# Let's change the step size.
stepSize = .01   # Try .1, .01, .001.

sol_t, sol_y = ode1( fun=weightf, t_span=tSpan, y0=yInit, h=stepSize )

plt.plot( sol_t, sol_y[0], label="position" )
plt.legend()
plt.xlabel( r"$t$" )
plt.ylabel( r"$y(t)$" )
plt.title( "Harmonic oscillator with y(0)={}, yDot(0)={}, h={}".format( yInit[0], yIn
print()
```

```
ode1 took 3142 steps
```



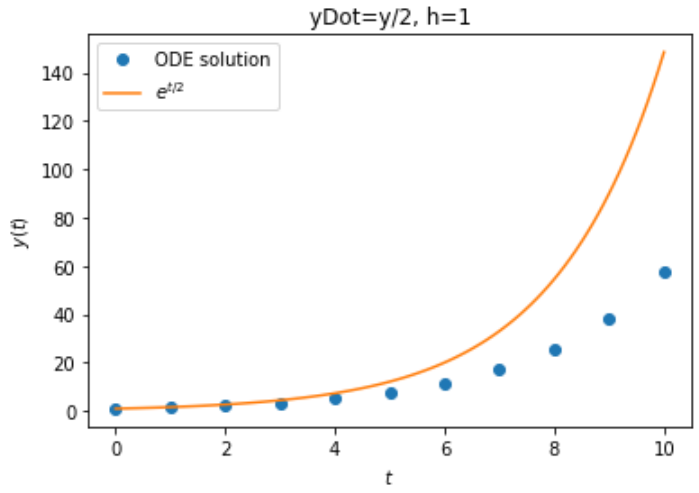
In [124]:
```
# Let's change the step size.
stepSize = .001   # Try .1, .01, .001.

sol_t, sol_y = ode1( fun=weightf, t_span=tSpan, y0=yInit, h=stepSize )

plt.plot( sol_t, sol_y[0], label="position" )
plt.legend()
plt.xlabel( r"$t$" )
plt.ylabel( r"$y(t)$" )
plt.title( "Harmonic oscillator with y(0)={}, yDot(0)={}, h={}".format( yInit[0], yIr
print()
```

```
ode1 took 31416 steps
```



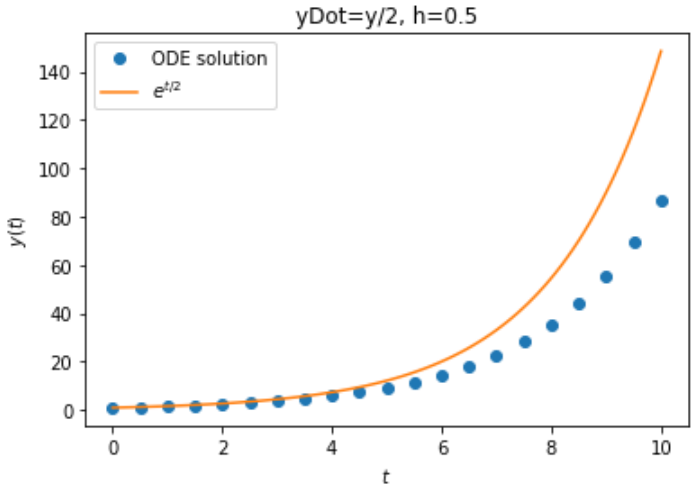
In [125]:
```
# Demo of ode1 with exp(t/2).
def f( t, y ):
    """
    Function to be integrated to solve an ODE or a system of ODEs.
    Input:
      t is a scalar time,
      y is a vector of variables.
    Output:
      yDot is the vector dy/dt.
    """
    yDot = y / 2
    return yDot

tSpan = (0, 10)
yInit = [1]
stepSize = 1 # Try 1, .5, .1, .01.

sol_t, sol_y = ode1( fun=f, t_span=tSpan, y0=yInit, h=stepSize )

plt.plot( sol_t, sol_y[0], 'o', label="ODE solution" )
tt = np.linspace( 0, 10, 100 )
plt.plot( tt, np.exp( tt / 2 ), label=r"$e^{t/2}$" )
plt.legend()
plt.xlabel( r"$t$" )
plt.ylabel( r"$y(t)$" )
plt.title( "yDot=y/2, h={}".format( stepSize ) )
print()
```

```
ode1 took 10 steps
```



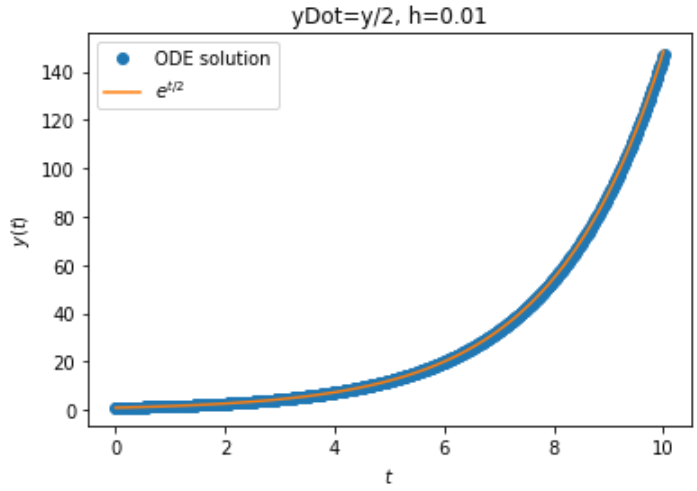
In [126]:
```
stepSize = .5 # Try 1, .5, .1, .01.

sol_t, sol_y = ode1( fun=f, t_span=tSpan, y0=yInit, h=stepSize )

plt.plot( sol_t, sol_y[0], 'o', label="ODE solution" )
tt = np.linspace( 0, 10, 100 )
plt.plot( tt, np.exp( tt / 2 ), label=r"$e^{t/2}$" )
plt.legend()
plt.xlabel( r"$t$" )
plt.ylabel( r"$y(t)$" )
plt.title( "yDot=y/2, h={}".format( stepSize ) )
print()
```

```
ode1 took 20 steps
```



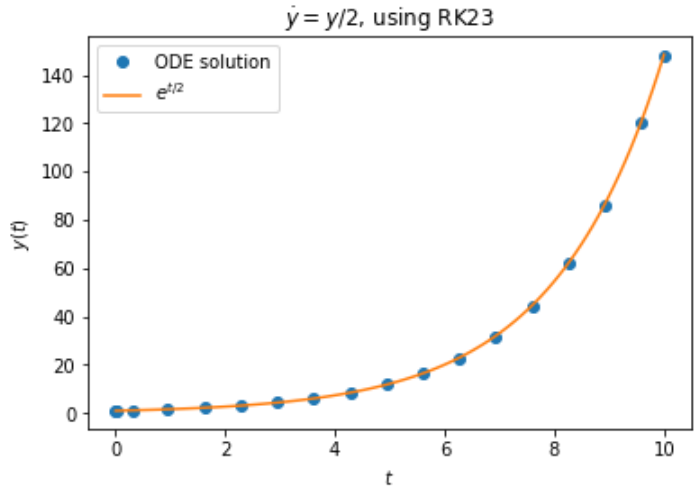
In [127]:
```
stepSize = .01 # Try 1, .5, .1, .01.

sol_t, sol_y = ode1( fun=f, t_span=tSpan, y0=yInit, h=stepSize )

plt.plot( sol_t, sol_y[0], 'o', label="ODE solution" )
tt = np.linspace( 0, 10, 100 )
plt.plot( tt, np.exp( tt / 2 ), label=r"$e^{t/2}$" )
plt.legend()
plt.xlabel( r"$t$" )
plt.ylabel( r"$y(t)$" )
plt.title( "yDot=y/2, h={}".format( stepSize ) )
print()
```

```
ode1 took 1001 steps
```



In [128]:
```
# Comparison with RK23 and exp(t/2).
sol = integrate.solve_ivp( fun=f, t_span=tSpan, y0=yInit, method="RK23" )
print( "RK23 took {} steps".format( len( sol.t ) ) )

plt.plot( sol.t, sol.y[0], 'o', label="ODE solution" )
tt = np.linspace( 0, 10, 100 )
plt.plot( tt, np.exp( tt / 2 ), label=r"$e^{t/2}$" )
plt.legend()
plt.xlabel( r"$t$" )
plt.ylabel( r"$y(t)$" )
plt.title( r"$\dot{y}=y/2$, using RK23" )
print()
```

```
RK23 took 18 steps
```

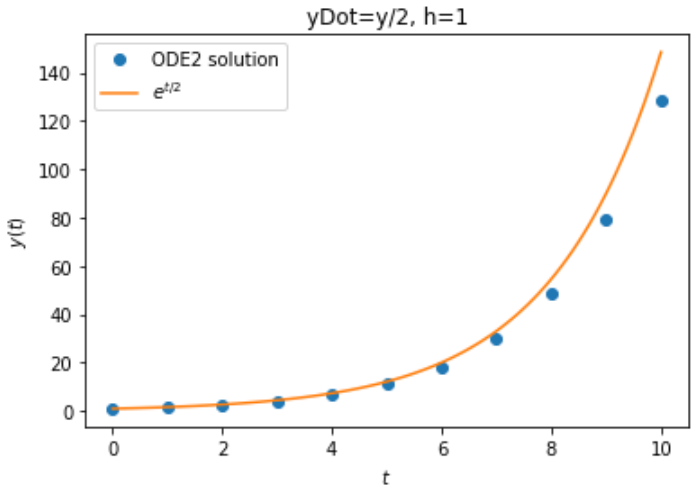


Midpoint Method

In [129]:
```
def ode2( fun, t_span, y0, h ):
    """
    Modified Euler algorithm that uses two slopes.
    ode2( fun, t_span, y0, h ) uses fixed step size h.
    """

    # First make the inputs into numpy arrays.
    t0 = np.array( t_span[0] ).reshape( 1 )
    tFinal = np.array( t_span[1] ).reshape( 1 )
    y0 = np.array( y0 ).reshape( len( y0 ), 1 )   # True column vector.

    # Initialize the list of solution points.
    sol_t = t0
    sol_y = y0

    step = 0
    t = t0
    y = y0
    while t < tFinal:
        s1 = np.array( fun( t, y ) )     # First slope.
        s2 = np.array( fun( t + h / 2, y + (h / 2) * s1 ) )  # Second slope, halfway
        y = y + h * s2
        t = t + h

        sol_t = np.concatenate( (sol_t, t) )
        sol_y = np.concatenate( (sol_y, y), axis=1 )
        step += 1
    print( "ode2 took", step, "steps" )
    return sol_t, sol_y
```

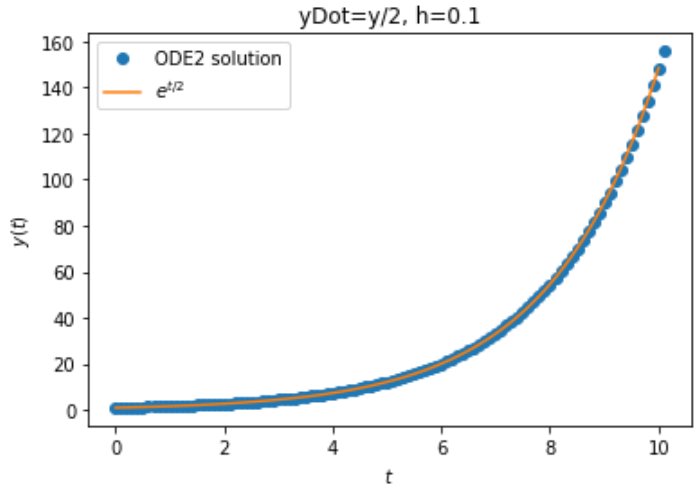
In [130]:
```
# Demo of ode2 with exp(t/2).
def f( t, y ):
    """
    Function to be integrated to solve an ODE or a system of ODEs.
    Input:
      t is a scalar time,
      y is a vector of variables.
    Output:
      yDot is the vector dy/dt.
    """
    yDot = y / 2
    return yDot

tSpan = (0, 10)
yInit = [1]
stepSize = 1 # Try 1, .1, .01.

sol_t, sol_y = ode2( fun=f, t_span=tSpan, y0=yInit, h=stepSize )

plt.plot( sol_t, sol_y[0], 'o', label="ODE2 solution" )
tt = np.linspace( 0, 10, 100 )
plt.plot( tt, np.exp( tt / 2 ), label=r"$e^{t/2}$" )
plt.legend()
plt.xlabel( r"$t$" )
plt.ylabel( r"$y(t)$" )
plt.title( "yDot=y/2, h={}".format( stepSize ) )
print()
```

```
ode2 took 10 steps
```



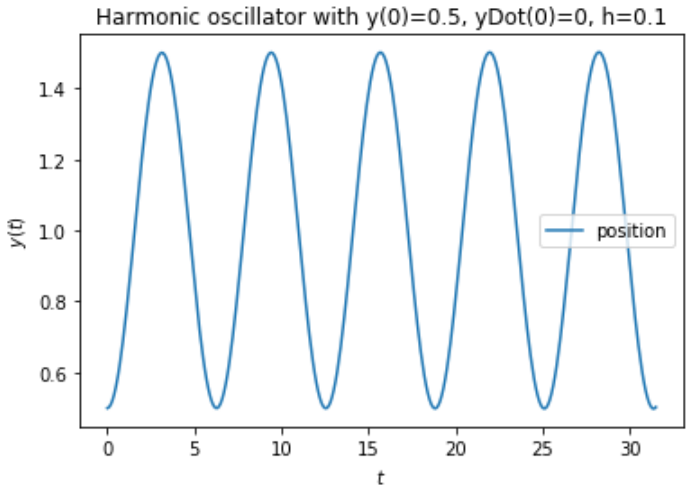
In [131]:
```
stepSize = .1 # Try 1, .1, .01.

sol_t, sol_y = ode2( fun=f, t_span=tSpan, y0=yInit, h=stepSize )

plt.plot( sol_t, sol_y[0], 'o', label="ODE2 solution" )
tt = np.linspace( 0, 10, 100 )
plt.plot( tt, np.exp( tt / 2 ), label=r"$e^{t/2}$" )
plt.legend()
plt.xlabel( r"$t$" )
plt.ylabel( r"$y(t)$" )
plt.title( "yDot=y/2, h={}".format( stepSize ) )
print()
```

```
ode2 took 101 steps
```



In [132]:
```
# Demo of ode1 with harmonic oscillator.
def weightf( t, y ):
    """
    Function to be integrated to solve a second-order ODE for a harmonic oscillator.
    Input:
      t is a scalar time,
      y is a vector of variables,
        in this case y = [position, 1st derivative]
    Output:
      yDot is the vector dy/dt,
        in this case ydot = [1st derivative, 2nd derivative]

    The second-order ODE being integrated is d^2 y / dt^2 = 1 - y.
    """
    yDot = [y[1] , 1 - y[0]]
    return yDot

tSpan = (0, 10 * np.pi)
yInit = [1/2, 0]
stepSize = .1   # Try .1, .01, .001.

sol_t, sol_y = ode2( fun=weightf, t_span=tSpan, y0=yInit, h=stepSize )

plt.plot( sol_t, sol_y[0], label="position" )
plt.legend()
plt.xlabel( r"$t$" )
plt.ylabel( r"$y(t)$" )
plt.title( "Harmonic oscillator with y(0)={}, yDot(0)={}, h={}".format( yInit[0], yIr
print()
```

```
ode2 took 315 steps
```



`RK23` (Same as `BS23` and `ode23tx.m`) Uses 3 New Slopes per Step

It belongs to the family of **Runge-Kutta** algorithms, which are *single-step* methods for solving ODEs.

The algorithm is due to Bogacki and Shampine.

When solving the problem of the bouncing weight,

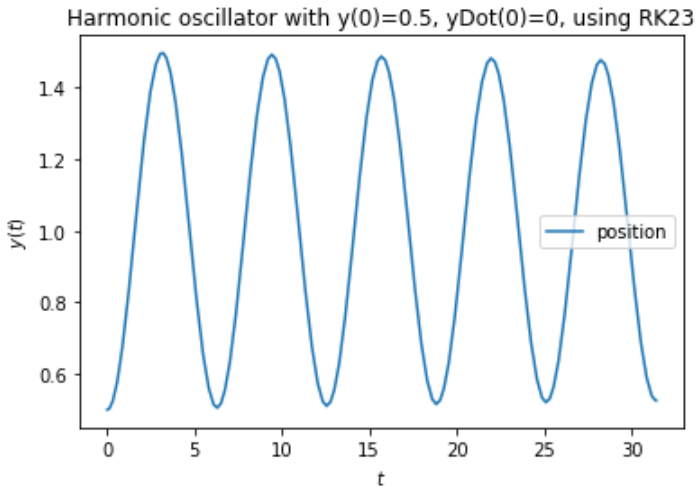
- `ode1` took 31416 steps (with the same number of function evaluations) to generate a good approximation to the solution,
- `ode2` took 315 steps or 630 function evaluations, and
- `rk23` took 104 steps or 312 function evaluations (3 slopes per step).

We can then say that, in general,

- `rk23` did twice as well as `ode2` in this problem, and
- `ode2` did ~50 times as well as `ode1` .

In [133]:
```
# Comparison with RK23 with harmonic oscillator
tspan = (0, 10*np.pi)
yinit = [1/2, 0]

sol = integrate.solve_ivp( fun=weightf, t_span=tSpan, y0=yInit, method="RK23" )
print('rk23 took %d steps' % len(sol.t))

plt.plot( sol.t, sol.y[0], label="position" )
plt.legend()
plt.xlabel( r"$t$" )
plt.ylabel( r"$y(t)$" )
plt.title( "Harmonic oscillator with y(0)={}, yDot(0)={}, using RK23".format( yInit[(
print()
```

```
rk23 took 105 steps
```



In [ ]:
```
```