Sparse eigenvalue problems with ARPACK

Introduction

ARPACK [1] is a Fortran package which provides routines for quickly finding a few eigenvalues/eigenvectors of large sparse matrices. In order to find these solutions, it requires only left-multiplication by the matrix in question. This operation is performed through a *reverse-communication* interface. The result of this structure is that ARPACK is able to find eigenvalues and eigenvectors of any linear function mapping a vector to a vector.

All of the functionality provided in ARPACK is contained within the two high-level interfaces **`scipy.sparse.linalg.eigs`** and **`scipy.sparse.linalg.eigsh`**. **`eigs`** provides interfaces for finding the eigenvalues/vectors of real or complex nonsymmetric square matrices, while **`eigsh`** provides interfaces for real-symmetric or complex-hermitian matrices.

Basic functionality

ARPACK can solve either standard eigenvalue problems of the form

$$A\mathbf{x} = \lambda\mathbf{x}$$

or general eigenvalue problems of the form

$$A\mathbf{x} = \lambda M\mathbf{x}.$$

The power of ARPACK is that it can compute only a specified subset of eigenvalue/eigenvector pairs. This is accomplished through the keyword `which`. The following values of `which` are available:

- `which = 'LM'` : Eigenvalues with largest magnitude (`eigs`, `eigsh`), that is, largest eigenvalues in the euclidean norm of complex numbers.
- `which = 'SM'` : Eigenvalues with smallest magnitude (`eigs`, `eigsh`), that is, smallest eigenvalues in the euclidean norm of complex numbers.
- `which = 'LR'` : Eigenvalues with largest real part (`eigs`).
- `which = 'SR'` : Eigenvalues with smallest real part (`eigs`).
- `which = 'LI'` : Eigenvalues with largest imaginary part (`eigs`).
- `which = 'SI'` : Eigenvalues with smallest imaginary part (`eigs`).
- `which = 'LA'` : Eigenvalues with largest algebraic value (`eigsh`), that is, largest eigenvalues inclusive of any negative sign.
- `which = 'SA'` : Eigenvalues with smallest algebraic value (`eigsh`), that is, smallest eigenvalues inclusive of any negative sign.
- `which = 'BE'` : Eigenvalues from both ends of the spectrum (`eigsh`).

Note that ARPACK is generally better at finding extremal eigenvalues, that is, eigenvalues with large magnitudes. In particular, using `which = 'SM'` may lead to slow execution time and/or anomalous results. A better approach is to use *shift-invert mode*.

Shift-invert mode

Shift-invert mode relies on the following observation. For the generalized eigenvalue problem

$$A\mathbf{x} = \lambda M\mathbf{x},$$

it can be shown that

$$(A - \sigma M)^{-1} M\mathbf{x} = \nu\mathbf{x},$$

where

$$\nu = \frac{1}{\lambda - \sigma}.$$

Examples

Imagine you'd like to find the smallest and largest eigenvalues and the corresponding eigenvectors for a large matrix. ARPACK can handle many forms of input: dense matrices ,such as `numpy.ndarray` instances, sparse matrices, such as `scipy.sparse.csr_matrix`, or a general linear operator derived from `scipy.sparse.linalg.LinearOperator`. For this example, for simplicity, we'll construct a symmetric, positive-definite matrix.

Search the docs ...

USER GUIDE

Special functions (scipy.special)

Integration (scipy.integrate)

Optimization (scipy.optimize)

Interpolation (scipy.interpolate)

Fourier Transforms (scipy.fft)

Signal Processing (scipy.signal)

Linear Algebra (scipy.linalg)

Sparse Arrays (scipy.sparse)

Sparse eigenvalue problems with ARPACK

Compressed Sparse Graph Routines (scipy.sparse.csgraph)

Spatial data structures and algorithms (scipy.spatial)

Statistics (scipy.stats)

Multidimensional image processing (scipy.ndimage)

File IO (scipy.io)

EXECUTABLE TUTORIALS

Interpolate transition guide

```
>>> import numpy as np
>>> from scipy.linalg import eig, eigh
>>> from scipy.sparse.linalg import eigs, eigsh
>>> np.set_printoptions(suppress=True)
>>> rng = np.random.default_rng()
>>>
>>> X = rng.random((100, 100)) - 0.5
>>> X = np.dot(X, X.T)  # create a symmetric matrix
```

We now have a symmetric matrix `X`, with which to test the routines. First, compute a standard eigenvalue decomposition using `eigh`:

```
>>> evals_all, evecs_all = eigh(X)
```

As the dimension of `X` grows, this routine becomes very slow. Especially, if only a few eigenvectors and eigenvalues are needed, `ARPACK` can be a better option. First let's compute the largest eigenvalues (`which = 'LM'`) of `X` and compare them to the known results:

```
>>> evals_large, evecs_large = eigsh(X, 3, which='LM')
>>> print(evals_all[-3:])
[29.22435321 30.05590784 30.58591252]
>>> print(evals_large)
[29.22435321 30.05590784 30.58591252]
>>> print(np.dot(evecs_large.T, evecs_all[:,-3:]))
array([[-1.  0.  0.],       # may vary (signs)
       [ 0.  1.  0.],
       [-0.  0. -1.]])
```

The results are as expected. ARPACK recovers the desired eigenvalues and they match the previously known results. Furthermore, the eigenvectors are orthogonal, as we'd expect. Now, let's attempt to solve for the eigenvalues with smallest magnitude:

```
>>> evals_small, evecs_small = eigsh(X, 3, which='SM')
Traceback (most recent call last):       # may vary (convergence)
...
scipy.sparse.linalg._eigen.arpack.arpack.ArpackNoConvergence:
ARPACK error -1: No convergence (1001 iterations, 0/3 eigenvectors converged)
```

Oops. We see that, as mentioned above, `ARPACK` is not quite as adept at finding small eigenvalues. There are a few ways this problem can be addressed. We could increase the tolerance (`tol`) to lead to faster convergence:

```
>>> evals_small, evecs_small = eigsh(X, 3, which='SM', tol=1E-2)
>>> evals_all[:3]
array([0.00053181, 0.00298319, 0.01387821])
>>> evals_small
array([0.00053181, 0.00298319, 0.01387821])
>>> np.dot(evecs_small.T, evecs_all[:,:3])
array([[ 0.99999999  0.00000024 -0.00000049],    # may vary (signs)
       [-0.00000023  0.99999999  0.00000056],
       [ 0.00000031 -0.00000037  0.99999852]])
```

This works, but we lose the precision in the results. Another option is to increase the maximum number of iterations (`maxiter`) from 1000 to 5000:

```
>>> evals_small, evecs_small = eigsh(X, 3, which='SM', maxiter=5000)
>>> evals_all[:3]
array([0.00053181, 0.00298319, 0.01387821])
>>> evals_small
array([0.00053181, 0.00298319, 0.01387821])
>>> np.dot(evecs_small.T, evecs_all[:,:3])
array([[ 1.  0.  0.],           # may vary (signs)
       [-0.  1.  0.],
       [ 0.  0. -1.]])
```

We get the results we'd hoped for, but the computation time is much longer. Fortunately, `ARPACK` contains a mode that allows a quick determination of non-external eigenvalues: *shift-invert mode*. As mentioned above, this mode involves transforming the eigenvalue problem to an equivalent problem with different eigenvalues. In this case, we hope to find eigenvalues near zero, so we'll choose `sigma = 0`. The transformed eigenvalues will then satisfy $\nu = 1/(\lambda - \sigma) = 1/\lambda$, so our small eigenvalues λ become large eigenvalues ν.

```
>>> evals_small, evecs_small = eigsh(X, 3, sigma=0, which='LM')
>>> evals_all[:3]
array([0.00053181, 0.00298319, 0.01387821])
>>> evals_small
array([0.00053181, 0.00298319, 0.01387821])
>>> np.dot(evecs_small.T, evecs_all[:,:3])
array([[ 1.  0.  0.],    # may vary (signs)
       [ 0. -1. -0.],
       [-0. -0.  1.]])
```

We get the results we were hoping for, with much less computational time. Note that the transformation from $\nu \to \lambda$ takes place entirely in the background. The user need not worry about the details.

The shift-invert mode provides more than just a fast way to obtain a few small eigenvalues. Say, you desire to find internal eigenvalues and eigenvectors, e.g., those nearest to $\lambda = 1$. Simply set `sigma = 1` and ARPACK will take care of the rest:

```
>>> evals_mid, evecs_mid = eigsh(X, 3, sigma=1, which='LM')
>>> i_sort = np.argsort(abs(1. / (1 - evals_all)))[-3:]
>>> evals_all[i_sort]
array([0.94164107, 1.05464515, 0.99090277])
>>> evals_mid
array([0.94164107, 0.99090277, 1.05464515])
>>> print(np.dot(evecs_mid.T, evecs_all[:,i_sort]))
array([[-0.  1.  0.],     # may vary (signs)
       [-0. -0.  1.],
       [ 1.  0.  0.]]
```

The eigenvalues come out in a different order, but they're all there. Note that the shift-invert mode requires the internal solution of a matrix inverse. This is taken care of automatically by `eigsh` and `eigs`, but the operation can also be specified by the user. See the docstring of `scipy.sparse.linalg.eigsh` and `scipy.sparse.linalg.eigs` for details.

Use of LinearOperator

We consider now the case where you'd like to avoid creating a dense matrix and use `scipy.sparse.linalg.LinearOperator` instead. Our first linear operator applies element-wise multiplication between the input vector and a vector $\mathbf{d}$ provided by the user to the operator itself. This operator mimics a diagonal matrix with the elements of $\mathbf{d}$ along the main diagonal and it has the main benefit that the forward and adjoint operations are simple element-wise multiplications other than matrix-vector multiplications. For a diagonal matrix, we expect the eigenvalues to be equal to the elements along the main diagonal, in this case $\mathbf{d}$. The eigenvalues and eigenvectors obtained with `eigsh` are compared to those obtained by using `eigh` when applied to the dense matrix:

```
>>> from scipy.sparse.linalg import LinearOperator
>>> class Diagonal(LinearOperator):
...     def __init__(self, diag, dtype='float32'):
...         self.diag = diag
...         self.shape = (len(self.diag), len(self.diag))
...         self.dtype = np.dtype(dtype)
...     def _matvec(self, x):
...         return self.diag*x
...     def _rmatvec(self, x):
...         return self.diag*x
```

```
>>> N = 100
>>> rng = np.random.default_rng()
>>> d = rng.normal(0, 1, N).astype(np.float64)
>>> D = np.diag(d)
>>> Dop = Diagonal(d, dtype=np.float64)
```

```
>>> evals_all, evecs_all = eigh(D)
>>> evals_large, evecs_large = eigsh(Dop, 3, which='LA', maxiter=1e3)
>>> evals_all[-3:]
array([1.53092498, 1.77243671, 2.00582508])
>>> evals_large
array([1.53092498, 1.77243671, 2.00582508])
>>> print(np.dot(evecs_large.T, evecs_all[:,-3:]))
array([[-1.  0.  0.],      # may vary (signs)
       [-0. -1.  0.],
       [ 0.  0. -1.]]
```

In this case, we have created a quick and easy `Diagonal` operator. The external library PyLops provides similar capabilities in the Diagonal operator, as well as several other operators.

Finally, we consider a linear operator that mimics the application of a first-derivative stencil. In this case, the operator is equivalent to a real nonsymmetric matrix. Once again, we compare the estimated eigenvalues and eigenvectors with those from a dense matrix that applies the same first derivative to an input signal:

```
>>> class FirstDerivative(LinearOperator):
...     def __init__(self, N, dtype='float32'):
...         self.N = N
...         self.shape = (self.N, self.N)
...         self.dtype = np.dtype(dtype)
...     def _matvec(self, x):
...         y = np.zeros(self.N, self.dtype)
...         y[1:-1] = (0.5*x[2:]-0.5*x[0:-2])
...         return y
...     def _rmatvec(self, x):
...         y = np.zeros(self.N, self.dtype)
...         y[0:-2] = y[0:-2] - (0.5*x[1:-1])
...         y[2:] = y[2:] + (0.5*x[1:-1])
...         return y
```

```
>>> N = 21
>>> D = np.diag(0.5*np.ones(N-1), k=1) - np.diag(0.5*np.ones(N-1), k=-1)
>>> D[0] = D[-1] = 0 # take away edge effects
>>> Dop = FirstDerivative(N, dtype=np.float64)
```

```
>>> evals_all, evecs_all = eig(D)
>>> evals_large, evecs_large = eigs(Dop, 4, which='LI')
>>> evals_all_imag = evals_all.imag
>>> isort_imag = np.argsort(np.abs(evals_all_imag))
>>> evals_all_imag = evals_all_imag[isort_imag]
>>> evals_large_imag = evals_large.imag
>>> isort_imag = np.argsort(np.abs(evals_large_imag))
>>> evals_large_imag = evals_large_imag[isort_imag]
>>> evals_all_imag[-4:]
array([-0.95105652, 0.95105652, -0.98768834, 0.98768834])
>>> evals_large_imag
array([0.95105652, -0.95105652, 0.98768834, -0.98768834])
```

Note that the eigenvalues of this operator are all imaginary. Moreover, the keyword `which='LI'` of **`scipy.sparse.linalg.eigs`** produces the eigenvalues with largest absolute imaginary part (both positive and negative). Again, a more advanced implementation of the first-derivative operator is available in the PyLops library under the name of FirstDerivative operator.

References

[1] https://github.com/opencollab/arpack-ng