Fourier Interpolation: iterpolation with $\sin$ and $\cos$

In [2]:

```
import numpy as np
import numpy.linalg as la
import matplotlib.pyplot as pt
%matplotlib inline
```

Let's fix the number of points and the number of functions as `n`.

Make sure `n` is odd.

In [3]:

```
n = 5

assert n % 2 == 1
```

In [4]:

```
x = np.linspace(0, 2*np.pi, n, endpoint=False)
```

Next, fix the values of k in $\cos(kx)$ as `kc` and in $sin(kx)$ as `ks` so that there are exactly n altogether.

We will look for coefficients `c` such that
$$f(x) = c_0 \cos kc_0 x + c_1 \sin ks_0 x + c_2 \cos kc_1 x + c_3 \sin ks_1 x + \dots$$
interpolates the data that we have

In [5]:

```
kc = np.arange(0, n//2 + 1, dtype=np.float64)
ks = np.arange(1, n//2 + 1, dtype=np.float64)
```

In [6]:

```
#keep
print(kc)
print(ks)
```

```
[ 0.  1.  2.]
[ 1.  2.]
```

Next, build the generalized Vandermonde matrix.

Make sure to order the matrix by increasing k:

In [7]:

```
V = np.zeros((n,n))
V[:, ::2] = np.cos(kc*x[:, np.newaxis])
V[:, 1::2] = np.sin(ks*x[:, np.newaxis])
V
```

Out[7]:

```
array([[ 1.        ,  0.        ,  1.        ,  0.        ,  1.        
],
       [ 1.        ,  0.95105652,  0.30901699,  0.58778525, -0.80901
699],
       [ 1.        ,  0.58778525, -0.80901699, -0.95105652,  0.30901
699],
       [ 1.        , -0.58778525, -0.80901699,  0.95105652,  0.30901
699],
       [ 1.        , -0.95105652,  0.30901699, -0.58778525, -0.80901
699]])
```

Notice the second column above is a clearly $\sin$ and the third column is clearly a $\cos$

now try to interpolate some functions

In [16]:

```
def f1(x):
    return x

def f2(x):
    return np.abs(x-np.pi)

def f3(x):
    return (x<=np.pi).astype(np.int32).astype(np.float64)

f = f3
```

Find the coefficients as `coeffs`:

In [17]:

```
coeffs = la.solve(V, f(x))
```

In [18]:

```
plot_x = np.linspace(0, 2*np.pi, 1000)
```

In [19]:

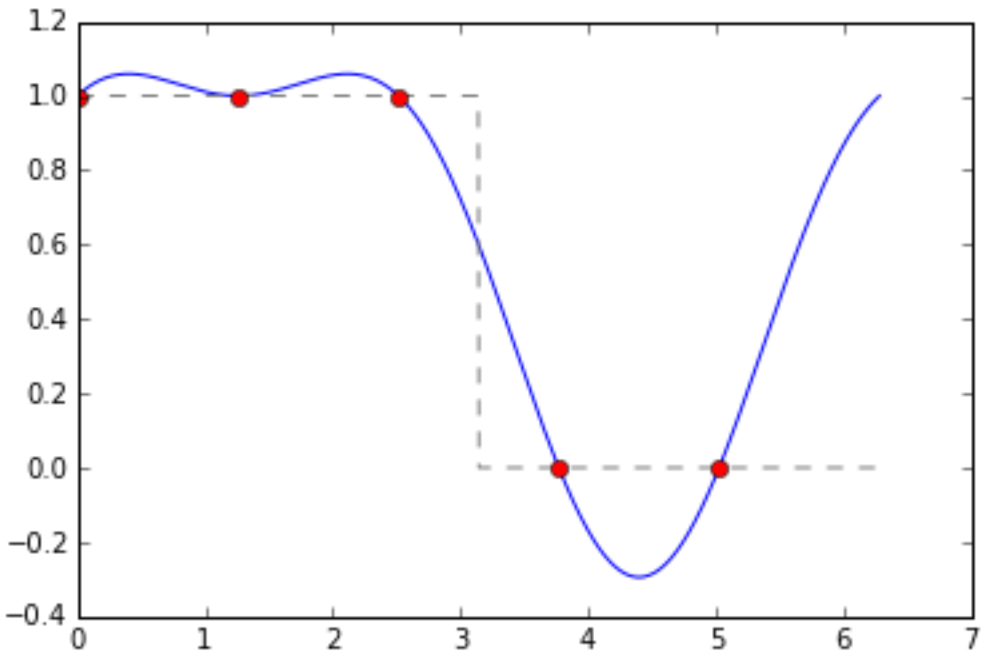
```
interpolant = 0 * plot_x
for i, k in enumerate(kc):
    interpolant += coeffs[2*i] * np.cos(k * plot_x)
for i, n in enumerate(ks):
    interpolant += coeffs[2*i+1] * np.sin(n * plot_x)
```

In [20]:

```
pt.plot(plot_x, interpolant)
pt.plot(plot_x, f(plot_x), "--", color="gray")
pt.plot(x, f(x), "or")
```

Out[20]:

```
[<matplotlib.lines.Line2D at 0x10e768320>]
```



- For $f(x) = x$, why does it do what it does at the interval end?
- What happens if we increase the number of points?
- What if the function is actually periodic (e.g. the function with `abs` above?)?
- What if the function has jumps?

In [21]:

```
# Answers
#
# * Because we're interpolating with periodic functions--so the interpolant is f
orced to be periodic.
# * We observe a distinct "overshoot". This overshoot is called "Gibbs phenomeno
n".
# * Periodic: no Gibbs at interval end.
# * Gibbs can also happen in the middle of the interval.
```

Computing the coefficients

In [22]:

```
B = V.T.dot(V)
```

In [23]:

```
B[np.abs(B)<1e-12] = 0
```

In [24]:

```
B
```

Out[24]:

```
array([[ 5. ,  0. ,  0. ,  0. ,  0. ],
       [ 0. ,  2.5,  0. ,  0. ,  0. ],
       [ 0. ,  0. ,  2.5,  0. ,  0. ],
       [ 0. ,  0. ,  0. ,  2.5,  0. ],
       [ 0. ,  0. ,  0. ,  0. ,  2.5]])
```

- What do you observe?
- How could this be useful for computing the inverse?
- What is the normalization?
- This is a pretty special matrix. What is the cost of computing Ax with it?
- The so-called Fast Fourier transform (https://en.wikipedia.org/wiki/Fast_Fourier_transform) (FFT) computes a version of Ax (with complex exponentials) in $O(n \log n)$!

In [25]:

```
# Answers:
#
# * V's columns are orthogonal. (though not normalized)
# * The transpose of $V$ (with appropriate normalization) its inverse. This make
s Fourier coefficients cheap to compute.
# * The normalization is n for the first entry, and n/2 for all the ones after t
hat.
# * Computing Ax costs n**2 operations.
```