

```
In [1]: 1 %matplotlib inline
2 from math import *
3 import numpy as np
4 import matplotlib.pyplot as plt
5 import matplotlib as mpl
```

Problem 1

(a)

```
In [2]: 1 # Polynomial
2 def poly(x, b):
3     z = b[0]
4     for i in range(1, n+1):
5         z *= x
6         z += b[i]
7     return z
8
9 # degree of the interpolant polynomial
10 n = 4
11
12 # Data points
13 x = np.array([1, 2, 3, 4, 5])
14 y = np.array([1, 1, 2, 6, 24])
15
16 # Solve b using Vandermonde Matrix (monomial basis)
17 V = np.vander(x)
18 b = np.linalg.solve(V, y)
19 print('The polynomial\'s coefficients for x^4, x^3, x^2, x, and the intercept are: ')
20 print(b)
```

The polynomial's coefficients for x^4 , x^3 , x^2 , x , and the intercept are:

```
[ 0.375      -3.41666667  11.625      -16.58333333   9.         ]
```

(b)

```
In [3]: 1 # Data points
2 x = np.array([1, 2, 3, 4, 5])
3 y = np.array([1, 1, 2, 6, 24])
4 y_log = np.log(np.array([1, 1, 2, 6, 24]))
5
6 # Solve b using monomial basis
7 V_log = np.vander(x)
8 b_log = np.linalg.solve(V, y_log)
9 print('The polynomial\'s(fitted on log(Gamma(n))) coefficients for x^4, x^3, x^2, x, and the intercept are: ')
10 print(b_log)
11
```

The polynomial's(fitted on $\log(\Gamma(n))$) coefficients for x^4 , x^3 , x^2 , x , and the intercept are:

```
[ 0.00707913 -0.11873828  0.88202509 -1.92109423  1.15072829]
```

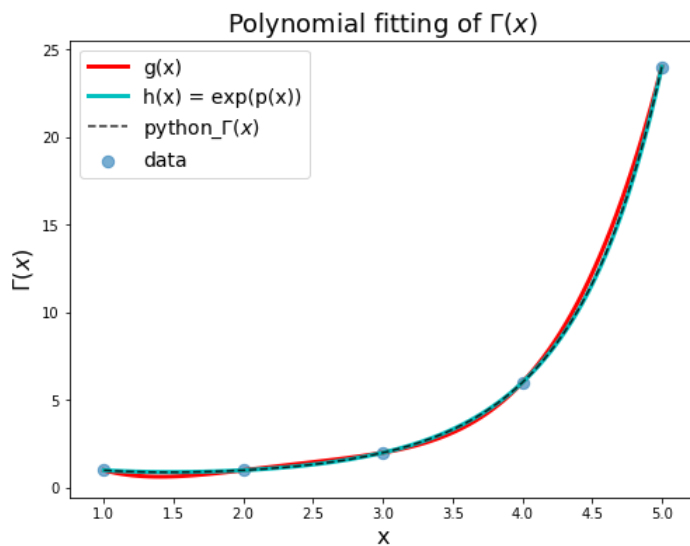
(c)

```

In [4]: 1 from scipy.special import gamma
2
3 # Plot result
4 fig, ax = plt.subplots(1, 1, figsize=(8, 6))
5 X = np.linspace(1, 5, 100)
6 g = np.array([poly(XX, b) for XX in X])
7
8 p = np.array([poly(XX, b_log) for XX in X])
9 h = np.exp(p)
10 python_g = np.array([gamma(XX) for XX in X])
11
12 ax.plot(X, g, 'r-', linewidth=3, label='g(x)')
13 ax.plot(X, h, 'c-', linewidth=3, label='h(x) = exp(p(x))')
14 ax.plot(X, python_g, 'k--', alpha=0.8, label='python_ $\Gamma(x)$')
15
16 # Plot the raw data points
17 ax.scatter(x=x, y=y, marker='o', alpha=0.6, s=70, label='data')
18
19 ax.set_xlabel('x', fontsize=16)
20 ax.set_ylabel(' $\Gamma(x)$', fontsize=16)
21 ax.set_title('Polynomial fitting of $\Gamma(x)$', fontsize=18)
22 ax.legend(fontsize=14)
23

```

Out[4]: <matplotlib.legend.Legend at 0x10d362f98>



(d)

```

In [5]: 1 # Compute maximum relative error of g(x) & h(x)
2 g_max_rel_e = np.max((np.abs(python_g - g)) / python_g)
3 g_max_rel_e_ind = np.argmax((np.abs(python_g - g)) / python_g)
4
5 h_max_rel_e = np.max((np.abs(python_g - h)) / python_g)
6 h_max_rel_e_ind = np.argmax((np.abs(python_g - h)) / python_g)
7
8 print('Max relative error between Gamma(x) and')
9 print('g(x) is {}, at x = {}'.format(g_max_rel_e, X[g_max_rel_e_ind]))
10 print('h(x) is {}, at x = {}'.format(h_max_rel_e, X[h_max_rel_e_ind]))

```

Max relative error between Gamma(x) and
g(x) is 0.2857257958185267, at x = 1.404040404040404
h(x) is 0.011516203377661278, at x = 1.3232323232323233

Problem 2

(a)

```
In [6]: 1 # Arrange the coefficients of the cubics in the matrix form
2 A = np.zeros((9, 9))
3 A[0:6, 1:7] = 2*np.eye(6)
4 B = np.zeros((9, 9))
5 B[1:7, 0:6] = 2*np.eye(6)
6
7 M = 8*np.eye(9)
8 M[7, 7] = 0
9 M[8, 8] = 0
10 M[0, 8] = 2
11 M[7, 0] = -2
12 M[7, 8] = -4
13 M[8, 6] = 2
14 M[8, 7] = 4
15
16 M += (A+B)
17 M
```

```
Out[6]: array([[ 8.,  2.,  0.,  0.,  0.,  0.,  0.,  0.,  2.],
 [ 2.,  8.,  2.,  0.,  0.,  0.,  0.,  0.,  0.],
 [ 0.,  2.,  8.,  2.,  0.,  0.,  0.,  0.,  0.],
 [ 0.,  0.,  2.,  8.,  2.,  0.,  0.,  0.,  0.],
 [ 0.,  0.,  0.,  2.,  8.,  2.,  0.,  0.,  0.],
 [ 0.,  0.,  0.,  0.,  2.,  8.,  2.,  0.,  0.],
 [ 0.,  0.,  0.,  0.,  0.,  2.,  8.,  0.,  0.],
 [-2.,  0.,  0.,  0.,  0.,  0.,  0.,  0., -4.],
 [ 0.,  0.,  0.,  0.,  0.,  0.,  2.,  4.,  0.]])
```

```
In [7]: 1 # Solve the coefficients of the cubics for the 8 splines
2 q0 = np.array([-6, 0, 0, 0, 0, 0, 0, 6, 0])
3 q1 = np.array([-6, 0, 0, 0, 0, 0, 0, -6, 0])
4 q2 = np.array([6, 0, -6, 0, 0, 0, 0, 0, 0])
5 q3 = np.array([0, 6, 0, -6, 0, 0, 0, 0, 0])
6 q4 = np.array([0, 0, 6, 0, -6, 0, 0, 0, 0])
7 q5 = np.array([0, 0, 0, 6, 0, -6, 0, 0, 0])
8 q6 = np.array([0, 0, 0, 0, 6, 0, -6, 0, 0])
9 q7 = np.array([0, 0, 0, 0, 0, 6, 0, 0, -6])
10 q8 = np.array([0, 0, 0, 0, 0, 0, 6, 0, 6])
11
12 qs = [q0, q1, q2, q3, q4, q5, q6, q7, q8]
13 ps = []
14 for q in qs:
15     p = np.linalg.solve(M, q)
16     ps.append(p)
```

In [8]:

```
1 # The 4 basis cubics used in class note
2 def c0(x):
3     return x*x*(3-2*x)
4 def c1(x):
5     return -x*x*(1-x)
6 def c2(x):
7     return (x-1)*(x-1)*x
8 def c3(x):
9     return 2*x*x*x-3*x*x+1
10
11 # Define the 9 splines s0, s1, ..., s8, using coefficients solved above
12 def s0(x):
13     params = ps[0]
14     a,b,c,d,e,f,g,h,i = params[0],params[1],params[2],params[3],params[4],params[5],params[6],params[7],params[8]
15     if x>=0 and x<1: return c3(x) + a*c1(x) + i*c2(x)
16     if x>=1 and x<2: return b*c1(x-1) + a*c2(x-1)
17     if x>=2 and x<3: return c*c1(x-2) + d*c2(x-2)
18     if x>=3 and x<4: return d*c1(x-3) + c*c2(x-3)
19     if x>=4 and x<5: return e*c1(x-4) + d*c2(x-4)
20     if x>=5 and x<6: return f*c1(x-5) + e*c2(x-5)
21     if x>=6 and x<7: return g*c1(x-6) + f*c2(x-6)
22     if x>=7 and x<=8: return h*c1(x-7) + g*c2(x-7)
23
24 def s1(x):
25     params = ps[1]
26     a,b,c,d,e,f,g,h,i = params[0],params[1],params[2],params[3],params[4],params[5],params[6],params[7],params[8]
27     if x>=0 and x<1: return c0(x) + a*c1(x) + i*c2(x)
28     if x>=1 and x<2: return c3(x-1) + b*c1(x-1) + a*c2(x-1)
29     if x>=2 and x<3: return c*c1(x-2) + d*c2(x-2)
30     if x>=3 and x<4: return d*c1(x-3) + c*c2(x-3)
31     if x>=4 and x<5: return e*c1(x-4) + d*c2(x-4)
32     if x>=5 and x<6: return f*c1(x-5) + e*c2(x-5)
33     if x>=6 and x<7: return g*c1(x-6) + f*c2(x-6)
34     if x>=7 and x<=8: return h*c1(x-7) + g*c2(x-7)
35
36 def s2(x):
37     params = ps[2]
38     a,b,c,d,e,f,g,h,i = params[0],params[1],params[2],params[3],params[4],params[5],params[6],params[7],params[8]
39     if x>=0 and x<1: return a*c1(x) + i*c2(x)
40     if x>=1 and x<2: return c0(x-1) + b*c1(x-1) + a*c2(x-1)
41     if x>=2 and x<3: return c3(x-2) + c*c1(x-2) + d*c2(x-2)
42     if x>=3 and x<4: return d*c1(x-3) + c*c2(x-3)
43     if x>=4 and x<5: return e*c1(x-4) + d*c2(x-4)
44     if x>=5 and x<6: return f*c1(x-5) + e*c2(x-5)
45     if x>=6 and x<7: return g*c1(x-6) + f*c2(x-6)
46     if x>=7 and x<=8: return h*c1(x-7) + g*c2(x-7)
47
48 def s3(x):
49     params = ps[3]
50     a,b,c,d,e,f,g,h,i = params[0],params[1],params[2],params[3],params[4],params[5],params[6],params[7],params[8]
51     if x>=0 and x<1: return a*c1(x) + i*c2(x)
52     if x>=1 and x<2: return b*c1(x-1) + a*c2(x-1)
53     if x>=2 and x<3: return c0(x-2) + c*c1(x-2) + d*c2(x-2)
54     if x>=3 and x<4: return c3(x-3) + d*c1(x-3) + c*c2(x-3)
55     if x>=4 and x<5: return e*c1(x-4) + d*c2(x-4)
56     if x>=5 and x<6: return f*c1(x-5) + e*c2(x-5)
57     if x>=6 and x<7: return g*c1(x-6) + f*c2(x-6)
58     if x>=7 and x<=8: return h*c1(x-7) + g*c2(x-7)
59
60 def s4(x):
61     params = ps[4]
62     a,b,c,d,e,f,g,h,i = params[0],params[1],params[2],params[3],params[4],params[5],params[6],params[7],params[8]
63     if x>=0 and x<1: return a*c1(x) + i*c2(x)
64     if x>=1 and x<2: return b*c1(x-1) + a*c2(x-1)
65     if x>=2 and x<3: return c*c1(x-2) + d*c2(x-2)
66     if x>=3 and x<4: return c0(x-3) + d*c1(x-3) + c*c2(x-3)
67     if x>=4 and x<5: return c3(x-4) + e*c1(x-4) + d*c2(x-4)
68     if x>=5 and x<6: return f*c1(x-5) + e*c2(x-5)
69     if x>=6 and x<7: return g*c1(x-6) + f*c2(x-6)
70     if x>=7 and x<=8: return h*c1(x-7) + g*c2(x-7)
71
72 def s5(x):
73     params = ps[5]
74     a,b,c,d,e,f,g,h,i = params[0],params[1],params[2],params[3],params[4],params[5],params[6],params[7],params[8]
75     if x>=0 and x<1: return a*c1(x) + i*c2(x)
76     if x>=1 and x<2: return b*c1(x-1) + a*c2(x-1)
77     if x>=2 and x<3: return c*c1(x-2) + d*c2(x-2)
78     if x>=3 and x<4: return d*c1(x-3) + c*c2(x-3)
79     if x>=4 and x<5: return c0(x-4) + e*c1(x-4) + d*c2(x-4)
80     if x>=5 and x<6: return c3(x-5) + f*c1(x-5) + e*c2(x-5)
81     if x>=6 and x<7: return g*c1(x-6) + f*c2(x-6)
```

```

82     if x>=7 and x<=8: return h*c1(x-7) + g*c2(x-7)
83
84 def s6(x):
85     params = ps[6]
86     a,b,c,d,e,f,g,h,i = params[0],params[1],params[2],params[3],params[4],params[5],params[6],params[7],param
87     if x>=0 and x<1: return a*c1(x) + i*c2(x)
88     if x>=1 and x<2: return b*c1(x-1) + a*c2(x-1)
89     if x>=2 and x<3: return c*c1(x-2) + d*c2(x-2)
90     if x>=3 and x<4: return d*c1(x-3) + c*c2(x-3)
91     if x>=4 and x<5: return e*c1(x-4) + d*c2(x-4)
92     if x>=5 and x<6: return c0(x-5) + f*c1(x-5) + e*c2(x-5)
93     if x>=6 and x<7: return c3(x-6) + g*c1(x-6) + f*c2(x-6)
94     if x>=7 and x<=8: return h*c1(x-7) + g*c2(x-7)
95
96 def s7(x):
97     params = ps[7]
98     a,b,c,d,e,f,g,h,i = params[0],params[1],params[2],params[3],params[4],params[5],params[6],params[7],param
99     if x>=0 and x<1: return a*c1(x) + i*c2(x)
100    if x>=1 and x<2: return b*c1(x-1) + a*c2(x-1)
101    if x>=2 and x<3: return c*c1(x-2) + d*c2(x-2)
102    if x>=3 and x<4: return d*c1(x-3) + c*c2(x-3)
103    if x>=4 and x<5: return e*c1(x-4) + d*c2(x-4)
104    if x>=5 and x<6: return f*c1(x-5) + e*c2(x-5)
105    if x>=6 and x<7: return c0(x-6) + g*c1(x-6) + f*c2(x-6)
106    if x>=7 and x<=8: return c3(x-7) + h*c1(x-7) + g*c2(x-7)
107
108 def s8(x):
109     params = ps[8]
110     a,b,c,d,e,f,g,h,i = params[0],params[1],params[2],params[3],params[4],params[5],params[6],params[7],param
111     if x>=0 and x<1: return a*c1(x) + i*c2(x)
112     if x>=1 and x<2: return b*c1(x-1) + a*c2(x-1)
113     if x>=2 and x<3: return c*c1(x-2) + d*c2(x-2)
114     if x>=3 and x<4: return d*c1(x-3) + c*c2(x-3)
115     if x>=4 and x<5: return e*c1(x-4) + d*c2(x-4)
116     if x>=5 and x<6: return f*c1(x-5) + e*c2(x-5)
117     if x>=6 and x<7: return g*c1(x-6) + f*c2(x-6)
118     if x>=7 and x<=8: return c0(x-7) + h*c1(x-7) + g*c2(x-7)
119
120 splines = [s0, s1, s2, s3, s4, s5, s6, s7, s8]
121
122 # Define the linear combination of the splines (S)
123 def spline_comb(x, b):
124     ret = 0
125     for bi, si in zip(b, splines):
126         ret += bi*si(x)
127     return ret

```

```

In [9]: 1 # Read in Xs, Ys from file
2 xs_1 = []
3 ys_1 = []
4 fl = './midterm_files/sdata1.txt'
5 with open(fl) as f:
6     lines = f.readlines()
7     for l in lines:
8         numbers = l.strip().split(' ')
9         xs_1.append(float(numbers[0]))
10        ys_1.append(float(numbers[1]))
11
12
13 # Arrange values of the 9 splines in matrix form
14 S = np.zeros((len(xs_1), 9))
15 for j, spline_k in enumerate(splines):
16     for i, x in enumerate(xs_1):
17         S[i, j] = spline_k(x)
18
19 # Linear least square fitting of the 9 splines on all data points (X, Y)
20 b1 = np.linalg.lstsq(S, np.array(ys_1))[0]
21 b1

```

```

Out[9]: array([ 0.51856191, -0.02703424, -0.87616979, -0.34414149,  0.60963002,
               -0.2735107 , -1.63220254, -2.94993774, -1.50771464])

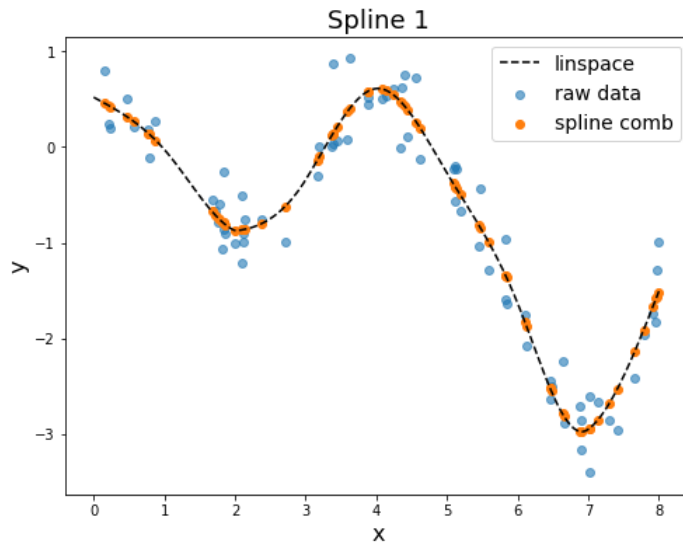
```

```

In [10]: 1 y_spline_comb = np.array([spline_comb(x, b1) for x in xs_1])
2
3 x_lin = np.linspace(0, 8, 200)
4 y_spline_comb_lin = np.array([spline_comb(x, b1) for x in list(x_lin)]) # Linear spaced continuous points
5
6
7 # Plot results
8 fig, ax = plt.subplots(1, 1, figsize=(8, 6))
9 ax.scatter(x=xs_1, y=ys_1, alpha=0.6, label='raw data')
10 ax.scatter(x=xs_1, y=y_spline_comb, label='spline comb')
11 ax.plot(x_lin, y_spline_comb_lin, 'k--', label='linspace')
12 ax.set_xlabel('x', fontsize=16)
13 ax.set_ylabel('y', fontsize=16)
14 ax.set_title('Spline 1', fontsize=18)
15 ax.legend(fontsize=14)

```

Out[10]: <matplotlib.legend.Legend at 0x10d840518>



(b)

```

In [11]: 1 # Read in Xs, Ys from file
2 xs_2 = []
3 ys_2 = []
4 f2 = './midterm_files/sdata2.txt'
5 with open(f2) as f:
6     lines = f.readlines()
7     for l in lines:
8         numbers = l.strip().split(' ')
9         xs_2.append(float(numbers[0]))
10        ys_2.append(float(numbers[1]))
11
12 # Arrange values of the 9 splines in matrix form
13 S2 = np.zeros((len(xs_2), 9))
14 for j, spline_k in enumerate(splines):
15     for i, x in enumerate(xs_2):
16         S2[i, j] = spline_k(x)
17
18 # Linear least square fitting of the 9 splines on all data points (X, Y)
19 b2 = np.linalg.lstsq(S, np.array(ys_2))[0]
20 b2

```

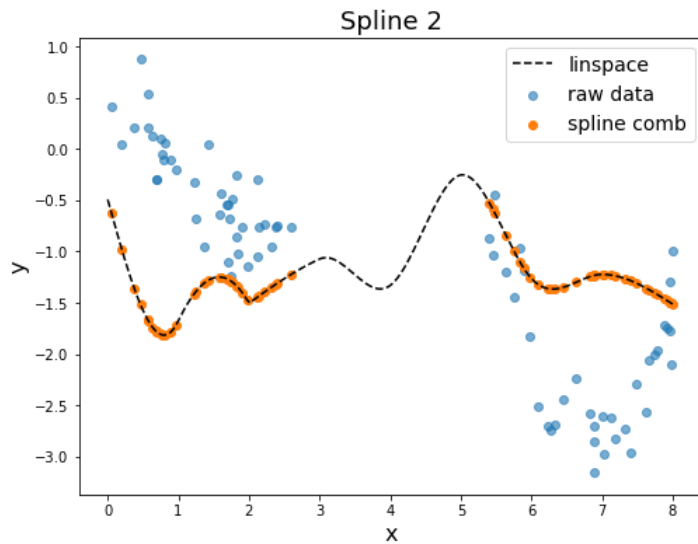
Out[11]: array([-0.49281047, -1.67616518, -1.50517194, -1.07335582, -1.33135505,
-0.25321862, -1.26349519, -1.22293233, -1.5130342])

```

In [12]: 1 y_spline_comb_2 = np.array([spline_comb(x, b2) for x in xs_2])
2
3 x_lin = np.linspace(0, 8, 200)
4 y_spline_comb_lin = np.array([spline_comb(x, b2) for x in list(x_lin)]) # Linear spaced continuous points
5
6 # Plot results
7 fig, ax = plt.subplots(1, 1, figsize=(8, 6))
8 ax.scatter(x=xs_2, y=ys_2, alpha=0.6, label='raw data')
9 ax.scatter(x=xs_2, y=y_spline_comb_2, label='spline comb')
10 ax.plot(x_lin, y_spline_comb_lin, 'k--', label='linspace')
11 ax.set_xlabel('x', fontsize=16)
12 ax.set_ylabel('y', fontsize=16)
13 ax.set_title('Spline 2', fontsize=18)
14 ax.legend(fontsize=14)

```

Out[12]: <matplotlib.legend.Legend at 0x10da38e10>



Problem 3

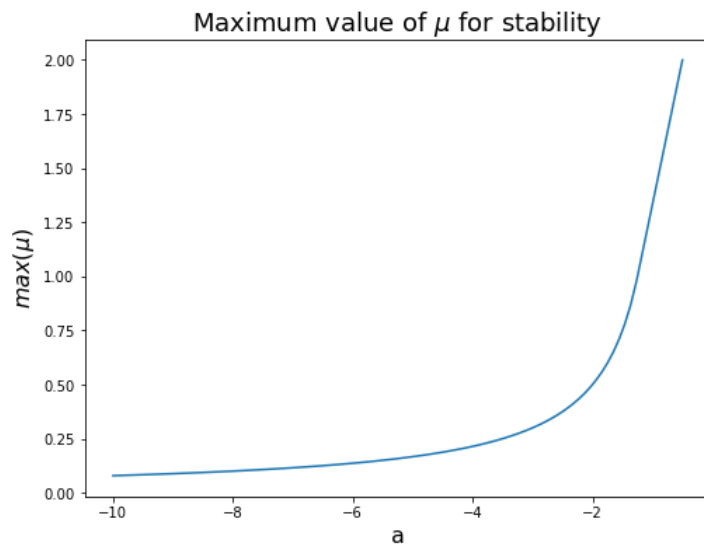
(f)

```

In [13]: 1 # Maximum value of mu as a function of a
          2 def mu(a):
          3     if a <= -5/4:
          4         return -3/(4*a+2)
          5     elif a <= -1/2:
          6         return (4*a+8)/3
          7     else:
          8         return 0
          9
         10 # Plot the maximum value of mu as a function of a
         11 a_lin = np.linspace(-10, -0.5, 100)
         12 mu_range = np.array([mu(a) for a in a_lin])
         13
         14 fig, ax = plt.subplots(1, 1, figsize=(8, 6))
         15 ax.plot(a_lin, mu_range)
         16 ax.set_xlabel('a', fontsize=16)
         17 ax.set_ylabel('$\max(\mu)$', fontsize=16)
         18 ax.set_title('Maximum value of $\mu$ for stability', fontsize=18)

```

Out[13]: <matplotlib.text.Text at 0x10db99080>



Problem 4

(c)

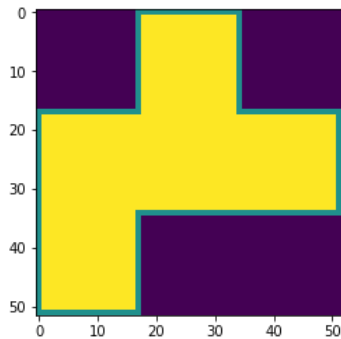

```

In [14]: 1 # Grid setup
2 m = 52
3 mm = m*m
4 h = 3.0/(m-1)
5 beta = 1.0/(5*h*h)
6 alpha = -8.0*beta
7 s = 1344
8
9 type_map = np.ones((m, m))
10
11 # mark exterior points
12 for j in range(0, 17):
13     for k in range(0, 17):
14         type_map[j, k] = -1
15 for j in range(0, 17):
16     for k in range(35, 52):
17         type_map[j, k] = -1
18 for j in range(35, 52):
19     for k in range(18, 52):
20         type_map[j, k] = -1
21
22 # mark boundary
23 for k in range(1, 18):
24     type_map[17, k] = 0
25 for j in range(0, 17):
26     type_map[j, 17] = 0
27 for k in range(18, 35):
28     type_map[0, k] = 0
29 for j in range(1, 18):
30     type_map[j, 34] = 0
31 for k in range(35, 52):
32     type_map[17, k] = 0
33 for j in range(18, 35):
34     type_map[j, 51] = 0
35 for k in range(17, 51):
36     type_map[34, k] = 0
37 for j in range(35, 52):
38     type_map[j, 17] = 0
39 for k in range(0, 17):
40     type_map[51, k] = 0
41 for j in range(17, 51):
42     type_map[j, 0] = 0
43
44 plt.title('90-degree Clockwise Rotated View of the Region Map\n', fontsize=16)
45 plt.imshow(type_map)

```

Out[14]: <matplotlib.image.AxesImage at 0x10d3eaf28>

90-degree Clockwise Rotated View of the Region Map



```

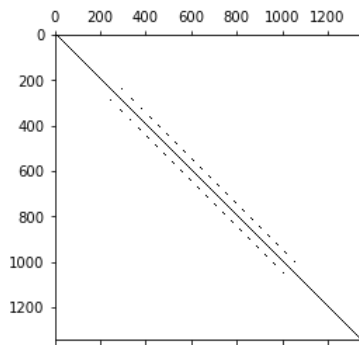
In [15]: 1 reindex_dic = {} # mapping 2D index to 1D
2 rev_reindex_dic = {} # mapping 1D index back to 2D
3 count = 0
4 for j in range(m):
5     for k in range(m):
6         if type_map[j, k] == 1:
7             reindex_dic[(j, k)] = count
8             rev_reindex_dic[count] = (j, k)
9             count += 1
10
11 print('The number of interior points in this region is {}'.format(count))

```

The number of interior points in this region is 1344

In [16]:

```
1 # Create derivative matrix and source term
2 A = np.zeros((s, s))
3 F = np.zeros(s)
4
5 ind = 0 # track the 1D indexing of interior points
6 for j in range(m):
7     x = j*h
8     for k in range(m):
9         if type_map[j, k] != 1:
10             continue
11
12         A[ind, reindex_dic[(j, k)]] = 1 - alpha
13
14         orth_neighbors = np.array([(j-2, k), (j-1, k), (j+1, k), (j+2, k),
15                                   (j, k-2), (j, k-1), (j, k+1), (j, k+2)])
16         for jn, kn in orth_neighbors:
17             if jn<0 or jn>=m or kn<0 or kn>=m : # neighbor is exterior: apply ghost node
18                 A[ind, reindex_dic[(j, k)]] += beta
19             elif type_map[jn, kn] == 1: # neighbor is interior
20                 A[ind, reindex_dic[(jn, kn)]] = -beta
21             elif type_map[jn, kn] == -1: # neighbor is exterior: apply ghost node
22                 A[ind, reindex_dic[(j, k)]] += beta
23
24         # Source term
25         y = k*h
26         F[ind] = 3 - (x-1.5)**2 + (y-1.5)**2
27
28         ind += 1
29
30 plt.spy(A)
31 plt.show()
```



In [17]:

```
1 # ||A - A.T||_{Frob.} == 0
2 np.linalg.norm(A - A.T, ord='fro')
```

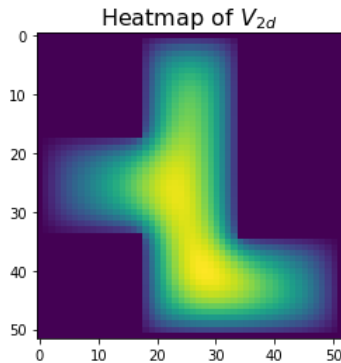
Out[17]: 0.0

In [18]:

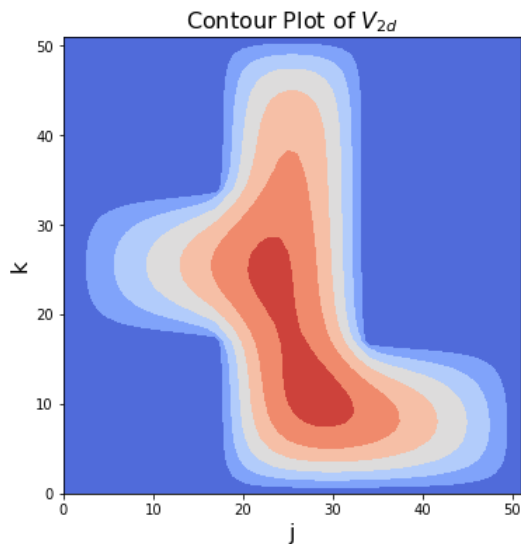
```
1 # Cholesky factorize A = L*L^T
2 L = np.linalg.cholesky(A)
3
4 # Solve for B: L*B = F
5 B = np.linalg.solve(L, F)
6
7 # Solve for V: L.T*V = B
8 V = np.linalg.solve(L.T, B)
```

```
In [19]: 1 # Recover 1D indexing to 2D
2 V_2d = np.zeros((m, m))
3 for i in range(s):
4     j, k = rev_reindex_dic[i]
5     V_2d[m-1-k, j] = V[i] # Correct array indexing to match (x, y) coordinates
6
7 plt.title('Heatmap of $V_{2d}$', fontsize=16)
8 plt.imshow(V_2d)
```

Out[19]: <matplotlib.image.AxesImage at 0x10da8d9e8>



```
In [20]: 1 # Plot the solution for V on the (x, y) plane
2 fig, ax = plt.subplots(1, 1, figsize=(8, 6))
3 ax.contourf(np.arange(m), np.arange(m)[::-1], V_2d, cmap=plt.cm.coolwarm)
4
5 ax.set_xlabel('j', fontsize=16)
6 ax.set_ylabel('k', fontsize=16)
7 plt.title('Contour Plot of $V_{2d}$', fontsize=16)
8 plt.gca().set_aspect('equal')
```



(d)

```
In [21]: 1 for i in range(s):
2     if V[i] == np.max(V):
3         j_max, k_max = rev_reindex_dic[i]
4         print('Maximun of V = {}'.format(V[i]))
5         print('At j = {}, k = {}'.format(j_max, k_max))
```

Maximun of V = 0.5147104874225262
At j = 28, k = 11

```
In [ ]: 1
```