In [261]:
```
%setup
```

```
pandas=pd, numpy=np
bokeh models, figure, layouts loaded
output directed to notebook
```

(https://bokeh.org) BokehJS 2.0.1 successfully loaded.

In [262]:
```
from bokeh.palettes import Category10
```

Momentum and Gradient Descent

We continue our discussion of the paper

> Why Momentum Really Works (https://distill.pub/2017/momentum)

See also the prior notes on the first part of this paper:

> Gradient Descent (./GradientDescent.html)

We continue with the analysis of the quadratic function $f(w) = -b^\intercal w + \frac{1}{2} w^\intercal A w$, with $A = Q\Lambda Q^\intercal$ and Λ diagonal with eigenvalues $\lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n$.

The diagonalized coordinates x are given by $x = Q(w - w_*)$ where $w_* = A^{-1}b$ is the minimum point of the function. For each of the diagonal coordinates x_i the quadratic function is

$$f(x_i) = \frac{1}{2}\lambda_i x_i^2$$

and the gradient $f'(x_i) = \lambda x_i$.

The iterative scheme for momentum (in the original w coordinates) involves introducing a sequence of variables $z_i^{(k)}$ and using the recursion:

$$\begin{aligned} z^{(k+1)} &= \beta z^{(k)} + \nabla f(w^{(k)}) \\ w^{(k+1)} &= w^{(k)} - \alpha z^{(k+1)} \end{aligned}$$

To interpret this, let $z^{(k+1)}$ represent the velocity at step k, so the second equation updates w by a step of duration α at speed $z^{(k+1)}$.

The first equation then updates the speed by adding a multiple of the prior speed to the gradient.

In the diagonal coordinates, these equations become

$$\begin{aligned} y^{(k+1)} &= \beta y^{(k)} + \lambda x^{(k)} \\ x^{(k+1)} &= x^{(k)} - \alpha y^{(k+1)} \end{aligned}$$

or in matrix form

$$\begin{bmatrix} y^{(k+1)} \\ x^{(k+1)} \end{bmatrix} = \begin{pmatrix} \beta & \lambda \\ -\alpha\beta & 1-\alpha\lambda \end{pmatrix} \begin{bmatrix} y^{(k)} \\ x^{(k)} \end{bmatrix}$$

This is a discrete dynamical system, or a Markov chain. λ is fixed, and we can adjust β and α to optimize the rate at which x goes to zero. As we expect, the dynamics of this are controlled by the eigenvalues of the matrix. In a diagonal basis, with eigenvalues r_1 and r_2, the iteration amounts to exponential growth r_i^n for each i. So we will have convergence provided both eigenvalues have absolute value < 1.

Working out when this happens is a little bit tricky as an algebra problem. Notice that the characteristic polynomial of this matrix is

$$T^2 - (\beta + 1 - \alpha\lambda)T + \beta.$$

Let's let $u = \alpha\lambda$. Then the discriminant of the polynomial is

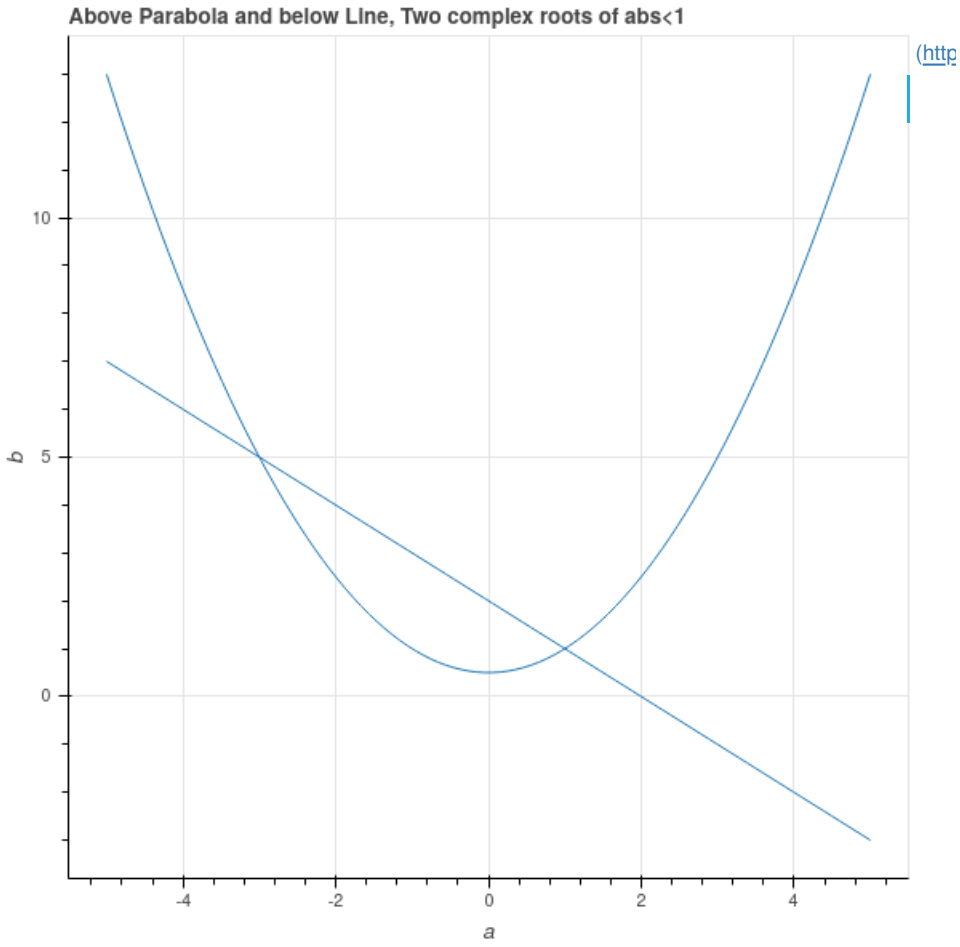
$$\begin{aligned} \Delta &= (\beta + 1 - u)^2 - 4\beta = \beta^2 + 1 + u^2 - 2u\beta - 2u + 2\beta - 4\beta \\ &= (\beta - u)^2 - 2(\beta + u) + 1 \end{aligned}$$

For further simplicity, let $a = \beta - u$ and $b = \beta + u$ so $\Delta = a^2 - 2b + 1$ and the characteristic polynomial is $T^2 - (a+1)T + (a+b)/2$.

Now if $\Delta < 0$, the two eigenvalues have the same absolute value and their product is $\beta = (a+b)/2$. So one condition is

$$\begin{aligned} b &> \frac{1}{2}(a^2 + 1) \\ b &< 2 - a \end{aligned}$$

In [263]:
```
a=np.linspace(-5,5,100)
b=(1/2)*(a**2+1)
f=figure()
f.line(x=a,y=b)
f.line(x=a,y=2-a)
f.title.text="Above Parabola and below Line, Two complex roots of abs<1"
f.xaxis.axis_label='a'
f.yaxis.axis_label='b'
show(f)
```



If $\Delta > 0$ there are two real roots and they are:

$$((a+1) \pm \sqrt{a^2 - 2b + 1})/2$$

The condition we want is then

$$(a+1) \pm \sqrt{a^2 - 2b + 1} < 2,$$

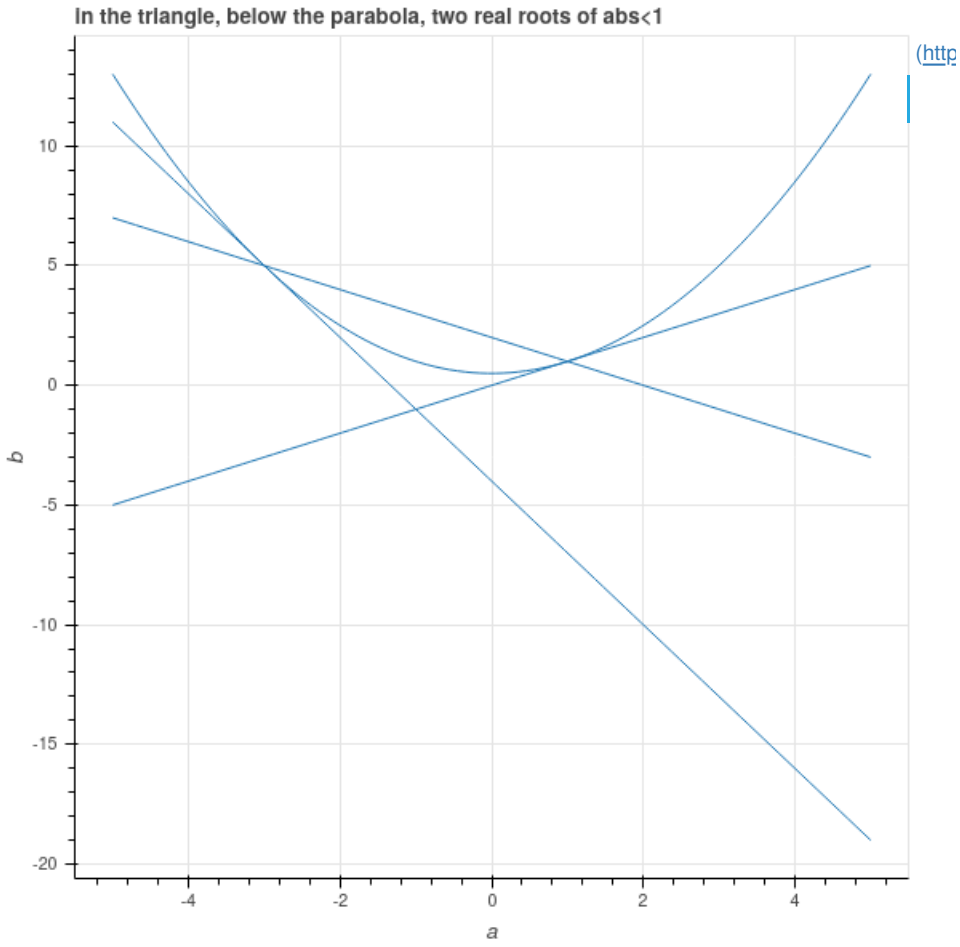
which yields $a < b$, and

$$(a+1) \pm \sqrt{a^2 - 2b + 1} > -2,$$

which yields

$$b > -4 - 3a$$

In [264]:
```
f=figure()
f.line(x=a,y=b)
f.line(x=a,y=2-a)
f.line(x=a,y=a)
f.line(x=a,y=-4-3*a)
f.xaxis.axis_label ='a'
f.yaxis.axis_label = 'b'
f.title.text ="in the triangle, below the parabola, two real roots of abs<1"
show(f)
```



In the picture above, the triangular region is the region of convergence for the algorithm. Translating back into the original coordinates yields the conditions:

$$0 \leq \beta < 1$$

and

$$0 < \alpha\lambda < 2\beta + 2.$$

Notice that we can know let $\alpha\lambda$ get nearly to 2 without losing convergence, by choosing β close to 1.

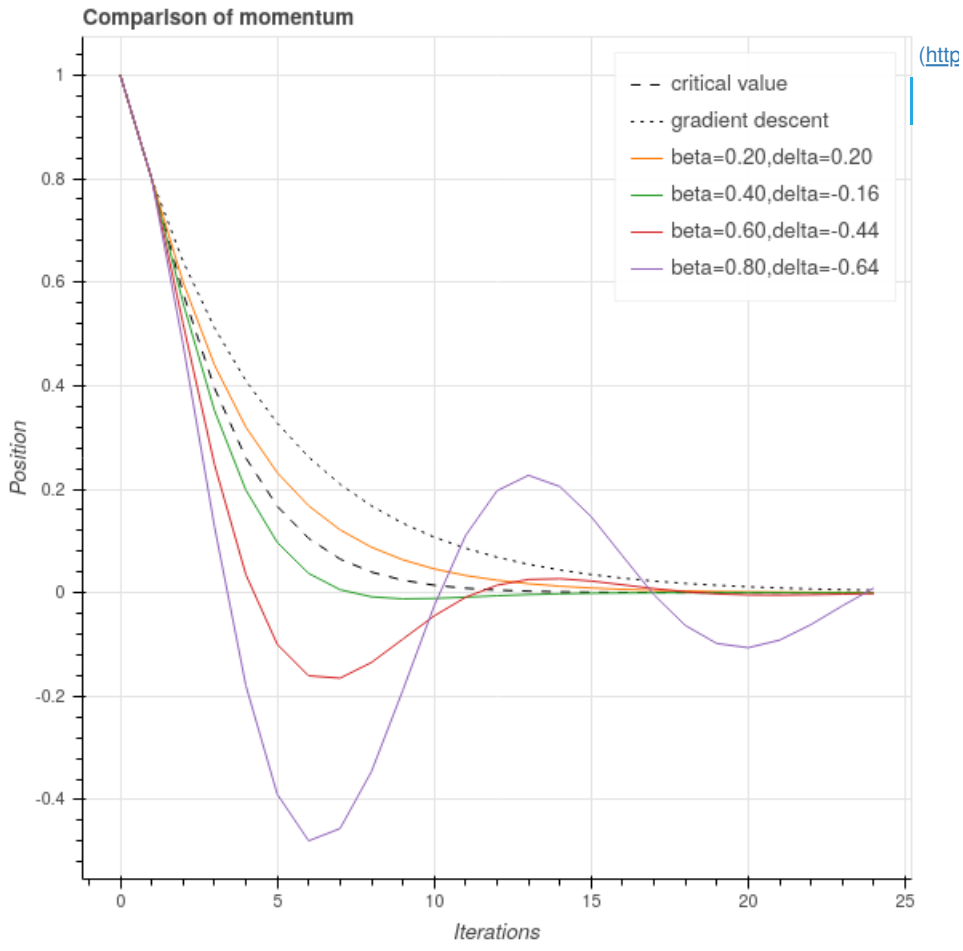
The optimal situation is when the two roots are real and equal. This happens when the discriminant $\Delta = 0$, or when $\beta = (1 - \sqrt{\alpha\lambda})^2$. The two eigenvalues, which control the rate of convergence, are $1 - \sqrt{\alpha\lambda}$. Comparing this to gradient descent (where the rate was $1 - \alpha\lambda$), we get an improvement!

Let's look at a particular example, with $\lambda = 0.2$, $\alpha = 1$, and varying β to see how the dynamics change.

In [265]:
```
def momentum(x0,alpha, beta, lam, N):
    '''return arrays xs, ys of length N containing x, y iterates'''
    ys=np.zeros(N)
    xs=np.zeros(N)
    xs[0]=x0
    for i in range(1,N):
        ys[i] = beta*ys[i-1]+lam*xs[i-1]
        xs[i] = xs[i-1] -alpha*ys[i]
    return xs, ys
```

In [266]:
```
f=figure()
f.xaxis.axis_label='Iterations'
f.yaxis.axis_label = 'Position'
f.title.text='Comparison of momentum'
N=25
alpha = 1
lam=.2
xcrit,ycrit = momentum(1,alpha,(1-np.sqrt(alpha*lam))**2,lam,N)

f.line(x=range(N),y=xcrit,line_dash='dashed',line_color='black',legend_label='cri
tical value')
xdescent,ydescent=momentum(1,alpha,0,lam,N)
f.line(x=range(N),y=xdescent,line_color='black',legend_label='gradient descent',l
ine_dash='dotted')
for i,beta in enumerate(np.arange(.2,1,.2)):
    delta = (beta+1-alpha*lam)**2-4*beta
    xs,ys=momentum(1,alpha,beta,lam,N)
    f.line(x=range(N),y=xs,color=Category10[10][i+1],legend_label='beta={:.2f},de
lta={:.2f}'.format(beta,delta))
show(f)
```



Now we have the problem of optimizing parameters for a collection of different eigenvectors $\lambda_1, \ldots, \lambda_n$. Following the same logic as before, the optimum α occurs when the two extremes give opposite rates:

$$(1 - \sqrt{\alpha\lambda_1}) = -(1 - \sqrt{\alpha\lambda_n})$$

or

$$\alpha = \left(\frac{2}{\sqrt{\lambda_1} + \sqrt{\lambda_n}} \right)^2.$$

For β, we need to maximize $(1 - \sqrt{\alpha\lambda})^2$ for this α, and that occurs when $\lambda = \lambda_1$ and

$$\beta = \left(\frac{\sqrt{\lambda_n} - \sqrt{\lambda_1}}{\sqrt{\lambda_n} + \sqrt{\lambda_1}} \right)^2$$

In our example from gradient descent, we have $f = .2x_1^2 + 2x_2^2 + 5x_3^2$, so the optimal values are as follows:

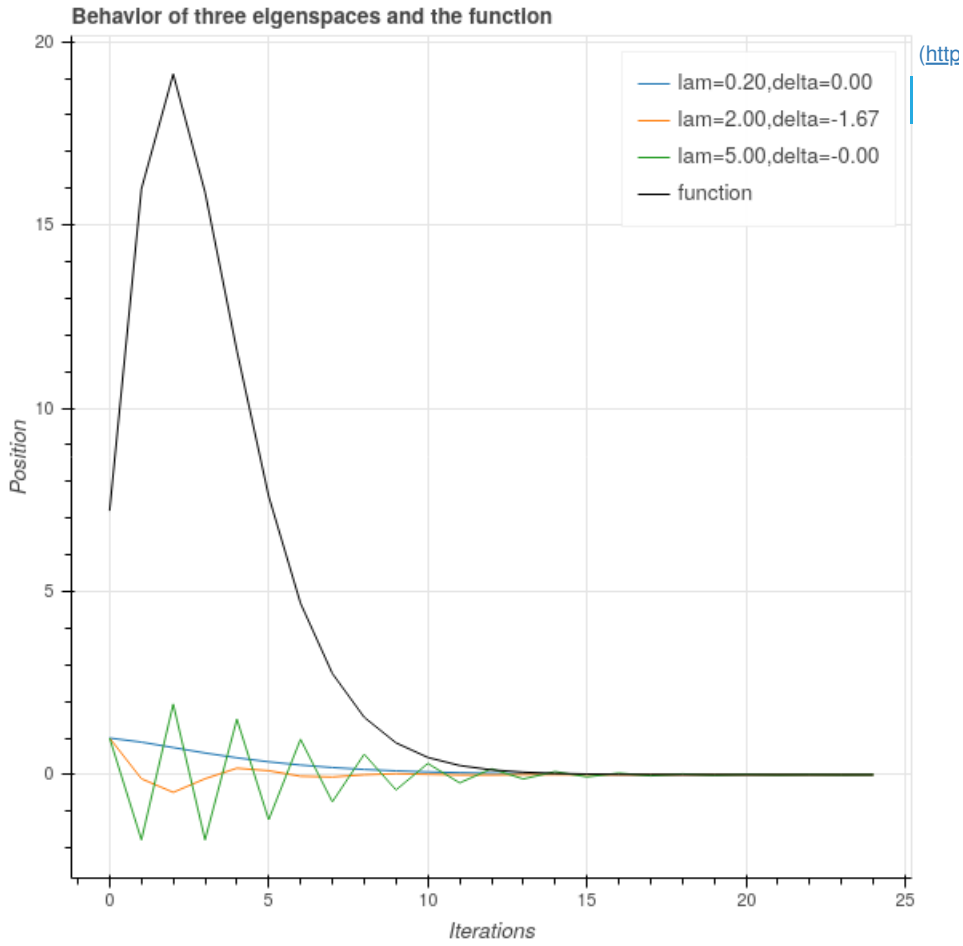
In [267]:
```
alpha = (2/(np.sqrt(.2)+np.sqrt(5)))**2
```

In [268]:
```
beta = ((np.sqrt(5)-np.sqrt(.2))/(np.sqrt(5)+np.sqrt(.2)))**2
```

In [269]:
```
alpha, beta
```

Out[269]:
```
(0.5555555555555556, 0.44444444444444453)
```

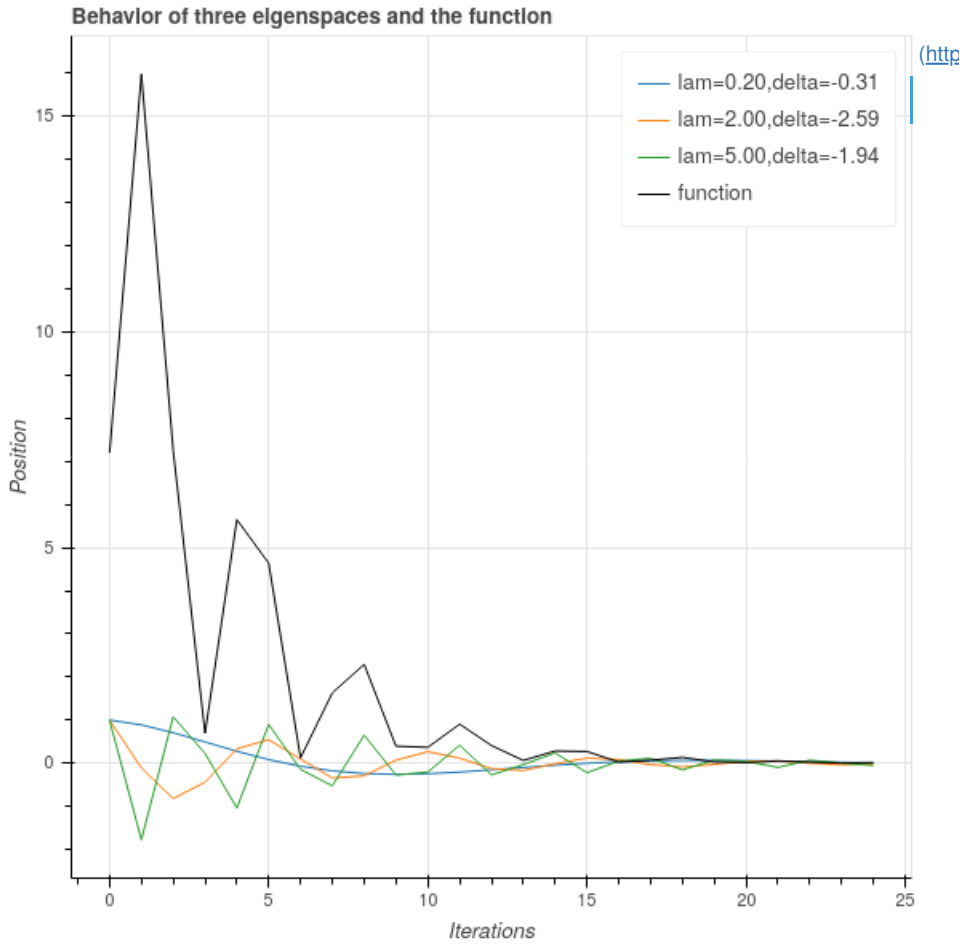
In [270]:
```
f=figure()
f.xaxis.axis_label='Iterations'
f.yaxis.axis_label = 'Position'
f.title.text='Behavior of three eigenspaces and the function'
alpha=.55555
beta=.444444
values = np.zeros(N)
for i,lam in enumerate([.2,2,5]):
    delta = (beta+1-alpha*lam)**2-4*beta
    xs,ys=momentum(1,alpha,beta,lam,N)
    values+= np.square(xs)*lam
    f.line(x=range(N),y=xs,color=Category10[3][i],legend_label='lam={:.2f},delta=
{:.2f}'.format(lam,delta))
f.line(x=range(N),y=values,color='black',legend_label='function')
show(f)
```



Note the oscillations. This is an artifact of the initial conditions, which aren't aligned with the eigenspaces. Since the matrix has repeated eigenvalues, need not act on the space like multiplication by λ it except in the limit.

Remark: Think about the Jordan normal form.

In [271]:
```
f=figure()
f.xaxis.axis_label='Iterations'
f.yaxis.axis_label = 'Position'
f.title.text='Behavior of three eigenspaces and the function'
alpha=.55555
beta=.75
values = np.zeros(N)
for i,lam in enumerate([.2,2,5]):
    delta = (beta+1-alpha*lam)**2-4*beta
    xs,ys=momentum(1,alpha,beta,lam,N)
    values+= np.square(xs)*lam
    f.line(x=range(N),y=xs,color=Category10[3][i],legend_label='lam={:.2f},delta=
{:.2f}'.format(lam,delta))
f.line(x=range(N),y=values,color='black',legend_label='function')
show(f)
```



In [ ]: