Enseñanza de la mecánica cuántica asistida por documentos interactivos. Reporte de proyecto

Semillero Física Teórica y Computacional

13 de diciembre de 2015

Resumen

El uso de las nuevas tecnologías en la educación permite integrar herramientas interactivas que permiten un acercamiento mas simple a la exploración de conceptos y de la interpretación de como los parámetros de un sistema afectan su comportamiento. Una manera de lograrlo se puede aproximar mediante la separación del uso de las herramientas en 3 etapas, encargadas respectivamente de la visualización e interacción, la solución numérica y la exploración del concepto físico. Para este reporte se presenta como caso de estudio la exploración de las soluciones de estados ligados 1D.

1. Introducción

El uso de las herramientas computacionales a permitido transformar las metodologías de enseñanza y acercar áreas comúnmente muy teóricas y de largos desarrollos matemáticos a procesos que pueden centrarse mucho mas en el concepto físico que en la matemática requerida [1, 2]. Su aprovechamiento en su fondo requiere del desarrollo de la física computacional, que constituye el desarrollo de métodos computacionales para la solución de problemas físicos [1].

La solución completa del problema físico no constituye la sola obtención de un resultado numérico sino la adecuada representación de los mismos y su interpretación, la cual mediante visualización se puede asistir el proceso.

Alrededor de estas iniciativas, y con especial enfoque en la mecánica cuántica existen acercamientos mediante *applets* [3], aplicaciones en html5 y javascript [4], códigos con métodos simples como diferencias finitas [5] y graficación tradicional y experiencias en HUBs con NanoHUBS [6]. Se presenta un gran uso del lenguaje python para este tipo de aplicaciones cuando son de uso local (no web) [1, 2] por su sencilla sintaxis y amplio soporte.

Mas recientemente, la integración de documentos interactivos de formatos abiertos (ipython notebooks y sage notebooks) [7, 8] y cerrados (maple worksheets) [9] ha permitido llevar a un solo ambiente, continuo y natural, las simulaciones y visualizaciones con sus respectivos materiales de apoyo, con especial interés aquellos formatos libres que brindan una mayor accesibilidad.

2. Método

Se desarrollo una metodología de documentos interactivos usando el formato abierto de Jupyter Notebook (anteriormente IPython Notebook) con integración de elementos interactivos *widgets* para la exploración de parámetros [10].

El documento interactivo se divide en 3 documentos acorde a la separación metodológica de la solución del problema físico, que se expone como parte de los tópicos básicos requeridos en física computacional en distintos currículos [11, 2]

- Visualización e interacción: Se exponen las necesidades de representación de los datos y la forma de realización, así como el uso de los controles para la interacción.
- Técnicas numéricas. Se expone las técnicas de aproximación numérica y su aplicación al contexto para la búsqueda de raíces y ecuaciones diferenciales con problema de frontera, y la metodología para contribuir a la estabilidad de las soluciones mediante el rescalamiento de valores por adimensionalización de la ecuación diferencial.
- Conceptualización física. Se presentan los elementos físicos habilitando una interacción amplia de los parámetros físicos y numéricos del fenómeno expuesto. Al respecto se explora el caso de estudio de estados ligados y superposición.

La accesibilidad del recurso es posible gracias al almacenamiento gratuito y publico ofrecido por el servidor git de github[1], y la ejecución en un servidor local posible mediante la instalación libre de jupyter notebooks o de su ejecución en servicios en linea gratuitos o por suscripción como SageMath [2] o Authorea[3].

Para una mayor discusión de los tópicos de la metodología, remítase a los notebooks, cuya versión estática se anexa a este documento.

[1] https://github.com/
[2] https://cloud.sagemath.com/
[3] https://www.authorea.com/

3. Resultados

Se genera satisfactoriamente la integración en documentos interactivos de formato abierto y gratuitos de desarrollos numéricos con estrategia de visualización e interacción de conceptos de la mecánica cuántica, con la presentación de casos de estudio de estados ligados y el concepto de paquete de onda.

La estrategia planteada permite crear un apoyo al desarrollo de un curso de mecánica cuántica sin dependencia de conocimientos previos en programación ni requerimientos específicos de instalaciones, salvo contar con un navegador web y acceso a internet. Siendo así una estrategia reproducible en otras instituciones sin limitación por requerimientos de una plataforma para la ejecución de los documentos interactivos.

Se logra independizar claramente las etapas en la estrategia de solución y así permitir el abordaje del problema físico con 3 niveles según el interés que se posea en las herramientas numéricas y de visualización, o si solo se trata del apoyo como recurso para la exploración de parámetros.

Una versión estática de los notebooks se anexa al reporte. Para su versión interactiva esta puede ser descargada del repositorio github Cuantica_Jupyter[4] del semillero[5] y cargarlos en un servidor jupyter (de instalación local o de un servicio en linea como sagemath, el cual es recomendado).

Para simulaciones con tiempos de ejecución altos, el servidor notebook retira la instancia tras un cierto lapso de tiempo. Para este tipo de casos se recomienda el uso directo de ipython o python.

4. Conclusiones

Es posible integrar de una manera natural las etapas de solución de un problema físico sin generar un solapamiento de un elemento con otro, que afecte el proceso de aprendizaje en medio del desarrollo y lectura de códigos de visualización y simulación, al separar estos en términos de la presentación con su respectiva documentación que permita no solo interactuar el documento de apoyo del curso original sino támbien una interacción con los nuevos elementos que se incorporan.

Al no ser complejos los elementos nuevos, se puede usar el documento conceptual como apoyo a los currículos tradicionales sin requerir de algún conocimiento de programación. Los elementos nuevos en la metodológia, permiten un acercamiento para cursos con un enfoque en computación y visualización científica, para el cual si es requerido un conocimiento previo en programación.

Es posible aumentar la accesibilidad de este recurso a un mayor publico gracias a ser un formato y herramienta abierta y gratuita, permitiendo una fácil descarga e instalación en caso de uso local o el uso de servicios en linea (gratuitos o por suscripción) que ofrecen servidos de jupyter notebook, dependiendo solo de conectividad a internet sin requerimientos especiales de hardware.

Para potenciales no simétricos y con amplias diferencias entre el potencial máximo y mínimo se recomienda usar instancias directas de python e ipython, debido al comportamiento como servidor de las instancias de Jupyter, que provocan la terminación del llamado al kernel tras un *timeout*, de aproximadamente minuto y medio. Tras este periodo se recomienda reiniciar el kernel (tanto si es de ejecución local como en linea).

Referencias

[1] J.F. Rojas, M.A. Morales, A. Rangel, and I. Torres. Física computacional: Una propuesta educativa. *Revista Mexicana de Física E*, 55(1):97–111, 2 2009.

[2] Rubin H. Landau, Manuel J. Paez, Cristian Bordeianu, and Sally Haerer. Making physics education more relevant and accesible via computation and etextbooks. *Computer Physics Communications*, 182:2071–2075, 2011.

[3] Mario Belloni, Wolfgang Christian, and Anne J. Cox. *Physlet Quantum Physics 2E*. 2015.

[4] University of Colorado Boulder. Phet interactive simulations: Quantum phenomena. Online. Last Accesed date 12-12-2015.

[5] R. Garcia, A. Zozulya, and J. Stickney. Matlab codes for teaching quantum physics: Part 1. page 7, 4 2007. arXiv:0704.1622 [physics.ed-ph].

[6] G. Klimeck. Nanohub.org tutorial: Education simulation tools. In *Nano Micro Engineered and Molecular Systems*, page 41. IEEE, IEEE, 1 2007.

[7] Fernando Perez and Brian E. Granger. An open source framework for interactive, collaborative and reproducible scientific computing and education. 2013.

[8] Halen Shen. Interactive notebooks: Sharing the code. *Nature*, 515:151–152, 11 2014.

[9] Marko Horbatsch. *Quantum Mechanics Using Maple ®*. Springer, 1995.

[10] Project Jupyter. Jupyter documentation, 2015. Last Accessed date 12-12-2015.

[11] Rubin H. Landau. Computational physics, a better model for physics education? *Computing in Science and Engineering*, 8(5):50–58, 9 2006.

[4] https://github.com/fisicatyc/Cuantica_Jupyter

[5] https://github.com/fisicatyc

Visualización e interacción

La visualización e interacción es un requerimiento actual para las nuevas metodologías de enseñanza, donde se busca un aprendizaje mucho más visual y que permita a traves de la experimentación el entendimiento de un fenomeno cuando se cambian ciertas condiciones iniciales.

La ubicación espacial y la manipulación de parametros en dicha experimentación se puede facilitar con herramientas como estas, que integran el uso de gráficos, animaciones y *widgets*. Este notebook, define los métodos de visualización e interacción que se usarán en otros notebooks, sobre la componente numérica y conceptual.

Esta separación se hace con el fin de distinguir claramente 3 componentes del proceso, y que faciliten la comprensión de la temática sin requerir que el usuario comprenda los 3 niveles (ya que el código es visible, y esto impactaría en el proceso de seguimiento del tema).

Funciones Matemáticas

Aunque no es parte de la visualización y de la interacción, el manejo de funciones matemáticas es requerido para estas etapas y las posteriores. Por lo que su definición es necesaria desde el principio para no ser redundante en requerir de múltiples invocaciones.

La evaluación de funciones matemáticas puede realizarse por medio del modulo `math` que hace parte de la biblioteca estandar de python, o con la biblioteca `numpy`. Para el conjunto limitado de funciones matemáticas que requerimos y con la premisa de no realizar de formas complejas nuestros códigos, las utilidades de `numpy` no serán necesarias y con `math` y el uso de listas será suficiente.

El motivo de tener pocos requerimientos de funciones matemáticas es por el uso de métodos numéricos y no de herramientas análiticas. La idea es mostrar como con esta metodología es posible analizar un conjunto mayor de problemas sin tener que profundizar en una gran cantidad de herramientas matemáticas y así no limitar la discusión de estos temas a conocimientos avanzados de matemáticas, y más bien depender de un conocimiento básico tanto de matemáticas como de programación para el desarrollo de los problemas, y permitir simplemente la interacción en caso de solo usar estos notebooks como un recurso para el estudio conceptual. Por este último fin, se busca que el notebook conceptual posea el mínimo de código, y este se lleve sobre los notebooks de técnicas numéricas y de visualización.

In [1]:
```
from math import sin, cos, tan, sqrt, log, exp, pi
```

El conjunto anterior de funciones solo se indica por mantener una referencia de funciones para cualquier ampliación que se desee realizar sobre este, y para su uso en la creación de potenciales arbitrarios, así como en los casos de ejemplificación con funciones análiticas o para fines de comparación de resultados.

Para la implementación (partiendo de un potencial dado numericamente), solo se requiere del uso de `sqrt`.

El modulo de `numpy` permitiría extender la aplicación de funciones matemáticas directamente sobre arreglos numéricos, y definir estos arreglos de una forma natural para la matemática, como equivalente a los vectores y matrices a traves de la clase `array`.

Interacción

Existen multiples mecanismos para interacción con los recursos digitales, definidos de forma casi estandar en su comportamiento a traves de distintas plataformas.

Dentro de la definición de los controles gráficos (*widgets*) incorporados en **Jupyter** en el modulo `ipywidgets`, encontramos los siguientes:

In [2]:
```
import ipywidgets
print(dir(ipywidgets))
```

```
['Accordion', 'BoundedFloatText', 'BoundedIntText', 'Box', 'Butt
on', 'CallbackDispatcher', 'Checkbox', 'Color', 'ColorPicker',
'CommInfo', 'Controller', 'DOMWidget', 'Dropdown', 'EventfulDic
t', 'EventfulList', 'FlexBox', 'FloatProgress', 'FloatRangeSlide
r', 'FloatSlider', 'FloatText', 'HBox', 'HTML', 'Image', 'IntPro
gress', 'IntRangeSlider', 'IntSlider', 'IntText', 'Latex', 'Outp
ut', 'PlaceProxy', 'Proxy', 'RadioButtons', 'Select', 'SelectMul
tiple', 'Tab', 'Text', 'Textarea', 'ToggleButton', 'ToggleButton
s', 'VBox', 'Valid', 'Widget', '__builtins__', '__cached__',
'__doc__', '__file__', '__loader__', '__name__', '__package__',
'__path__', '__spec__', '__version__', '_handle_ipython', '_vers
ion', 'eventful', 'find_static_assets', 'fixed', 'get_ipython',
'handle_kernel', 'interact', 'interact_manual', 'interaction',
'interactive', 'jsdlink', 'jslink', 'load_ipython_extension', 'o
s', 'register', 'register_comm_target', 'trait_types', 'versio
n_info', 'widget', 'widget_bool', 'widget_box', 'widget_button',
'widget_color', 'widget_controller', 'widget_float', 'widget_ima
ge', 'widget_int', 'widget_link', 'widget_output', 'widget_selec
tion', 'widget_selectioncontainer', 'widget_serialization', 'wid
get_string', 'widgets']
```

Para nuestro uso, serán de uso principal:

- Interacciones: Son mecanismos automaticos para crear controles y asociarlos a una función. `interact`, `interactive`.
- Deslizadores: Los hay especificos para tipos de datos, y estos son `IntSlider` y `FloatSlider`.
- Botones: Elementos que permiten ejecutar una acción al presionarlos, `Button`.
- Texto: Permiten el ingreso de texto arbitrario y asociar la ejecución de una acción a su ingreso. `Text`.
- Contenedores: Permiten agrupar en un solo objeto/vista varios controles. Uno de ellos es

In [3]:
```
from ipywidgets import interact, interactive, fixed, IntSlider,
FloatSlider, Button, Text, Box
```

Entre estos controles que se usan, a veces es necesario crear dependencias de sus rangos respecto al rango o propiedad de otro control. Para este fin usamos la función `link` del modulo `traitlets`. En este modulo se encuentran otras funciones utiles para manipulación de los controles gráficos.

In [4]:
```
from traitlets import link
```

Tambien es necesario el uso de elementos que permitan el formato del documento y visualización de elementos y texto enriquecido, fuera de lo posible con texto plano a punta de `print` o con las capacidades de *MarkDown* (https://daringfireball.net/projects/markdown/) (Nativo (http://jupyter-notebook.readthedocs.org/en/latest/examples/Notebook/rstversions/Working%20With%20Markdown%2
o con extensión (https://github.com/ipython-contrib/IPython-notebook-extensions/wiki/python-markdown)). Para esto se puede extender el uso métodos para renderizado HTML y LaTeX

In [5]:
```
from IPython.display import clear_output, display, HTML, Latex,
Markdown, Math
```

Visualización

Por visualización entendemos las estrategias de representación gráfica de la información, resultados o modelos. Facilita la lectura rápida de datos mediante codificaciones de colores así como la ubicación espacial de los mismos. La representación gráfica no tiene por que ser estatica, y es ahí donde las animaciones nos permiten representar las variaciones temporales de un sistema de una forma más natural (no como un grafico respecto a un eje de tiempo, sino vivenciando un grafico evolucionando en el tiempo).

Para este fin es posible usar diversas bibliotecas existentes en python (en sus versiones 2 y 3), siendo la más común de ellas y robusta, la biblioteca de Matplotlib (http://matplotlib.org/). En el contexto moderno de los navegadores web, es posible integrar de una forma más natural bibliotecas que realizan el almacenamiento de los gráficos en formatos nativos para la web, como lo es el formato de intercambio de datos JSON (http://www.json.org/), facilitando su interacción en el navegador mediante llamados a javascript (https://en.wikipedia.org/wiki/JavaScript).

Así, podemos establecer preferencias, como Matplotlib para uso estatico principalmente o para uso local, mientras que para interacción web, usar bibliotecas como Bokeh (http://bokeh.pydata.org/en/latest/).

Para este caso, sin profundidad en la interacción web, se usara Matplotlib.

In [6]:
```
%matplotlib inline
import matplotlib.pyplot as plt
```

Para indicar la graficación no interactiva embebida en el documento usamos la sguiente linea

```
%matplotlib inline
```

En caso de requerir una forma interactiva embebida, se usa la linea

```
%matplotlib notebook
```

Para nuestro uso básico, todo lo necesario para gráficación se encuentra en el modulo pyplot de Matplotlib. Con él podemos realizar cuadriculas, trazos de curvas de diversos estilos, modificación de ejes, leyendas, adición de anotaciones en el gráfico y llenado de formas (coloreado entre curvas). Pueden consultarse ejemplos de referencia en la galería (http://matplotlib.org/gallery.html) de Matplotlib y en la lista de ejemplos (http://matplotlib.org/examples/index.html) de la página oficial.

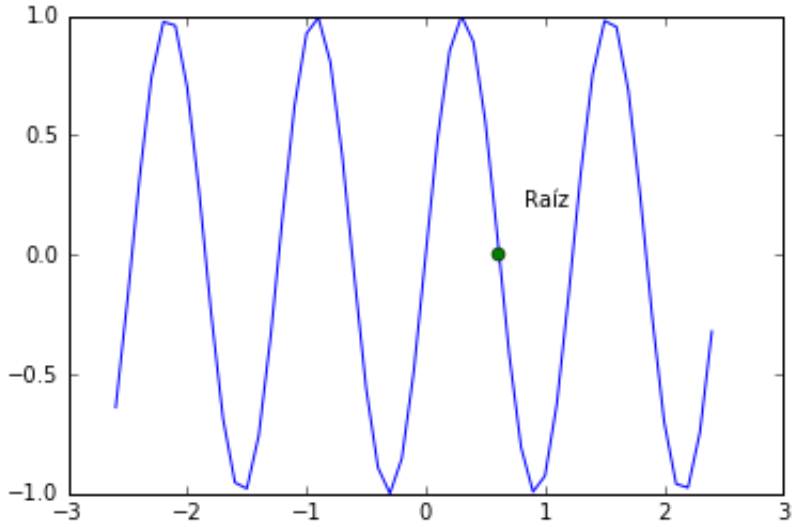
Graficación de funciones

En general nuestro ideal es poder graficar funciones que son representadas por arreglos numéricos. Las funciones continuas en su representación algebraica de discretizan, y es el conjunto de puntos interpolado lo que se ilustra. Antes de discretizar, es conveniente convertir nuestra función en una función evaluable, y asociar la dependencia solo a una variable (para nuestro caso que es 1D).

El proceso de interpolación mencionado se realiza por el paquete de graficación y nosotros solo debemos indicar los puntos que pertenecen a la función.

In [7]:
```
def discretizar(funcion, a, b, n):
    dx = (b-a)/n
    x = [a + i*dx for i in range(n+1)]
    y = [funcion(i) for i in x]
    return x, y

def graficar_funcion(x, f):
    plt.plot(x, f, '-')

def graficar_punto_texto(x, f, texto):
    plt.plot(x, f, 'o')
    plt.text(x+.2, f+.2, texto)
```

In [8]:
```
def int_raiz_sin(a:(-5.,0., .2), b:(0., 5., .2), k:(0.2, 10., .1), n:(1, 100, 1), N:(0, 10, 1)):
    f = lambda x: sin(k*x)
    x, y = discretizar(f, a, b, n)
    r = pi*(N + int(a*k/pi))/k
    graficar_funcion(x, y)
    graficar_punto_texto(r, 0, 'Raíz')
    plt.show()

interact(int_raiz_sin)
```



Out[8]: <function __main__.int_raiz_sin>

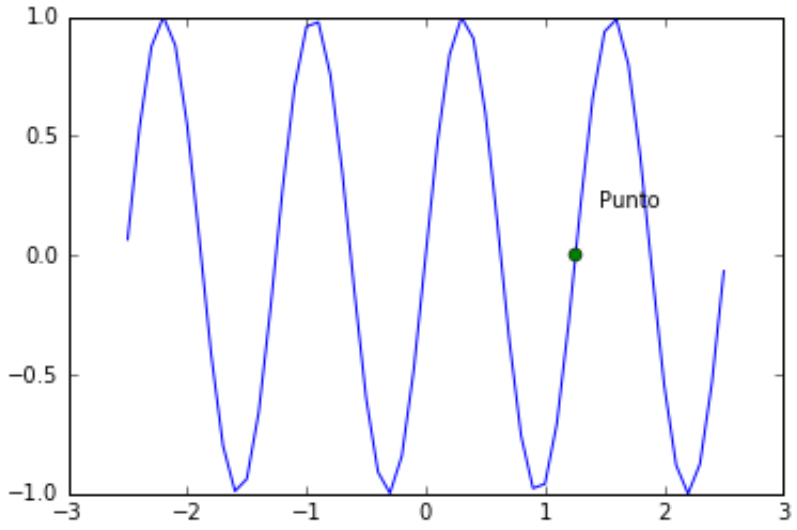
El bloque anterior de código ilustra el uso de `interact` como mecanismo para crear controles automaticos que se apliquen a la ejecución de una función. Este permite crear de una forma simple las interacciones cuando no se requiere de personalizar mucho, ni vincular controles y se desea una ejecución automatica con cada variación de parametros. En caso de querer recuperar los valores especificos de los parametros para posterior manipulación se recomienda el uso de `interactive` o del uso explicito de los controles.

A pesar de la facilidad que ofrece `interact` e `interactive` al generar los controles automaticos, esto es poco conveniente cuando se trata de ejecuciones que toman tiempos significativos (que para escalas de una interacción favorable, un tiempo significativo son aquellos mayores a un segundo), ya que cada variación de parametros independiente, o sea, cada deslizador en este caso, al cambiar produce una nueva ejecución, y las nuevas variaciones de parametros quedan en espera hasta terminar las ejecuciones de las variaciones individuales anteriores.

Es por esto, que puede ser conveniente definir una interacción donde los controles la unica acción que posean es la variación y almacenamiento de valores de los parametros, y sea otro control adicional el designado para indicar el momento de actualizar parametros y ejecutar.

El ejemplo anterior se puede construir usando `FloatSlider`, `IntSlider`, `Button`, `Text`, `Box` y `display`.

In [9]:
```
def raiz_sin(a, b, k, n, N, texto):
    f = lambda x: sin(k*x)
    x, y = discretizar(f, a, b, n)
    r = pi*(N + int(a*k/pi))/k
    graficar_funcion(x, y)
    graficar_punto_texto(r, 0, texto)

a = FloatSlider(value= -2.5, min=-5., max= 0., step= .2, descrip
tion='a')
b= FloatSlider(value = 2.5, min=0., max= 5., step=.2, descriptio
n='b')
k= FloatSlider(value = 5., min=0.2, max=10., step=.1, descriptio
n='k')
n= IntSlider(value= 50, min=1, max= 100, step=1, descriptio
n='n')
N= IntSlider(value=5, min=0, max=10, step=1, description='N')
texto = Text(value='Raíz', description='Texto punto')
Boton_graficar = Button(description='Graficar')

def click_graficar(boton):
    clear_output(wait=True)
    raiz_sin(a.value, b.value, k.value, n.value, N.value, text
o.value)
    plt.show()

display(a, b, k, n, N, texto, Boton_graficar)
Boton_graficar.on_click(click_graficar)
```



Graficación de potenciales

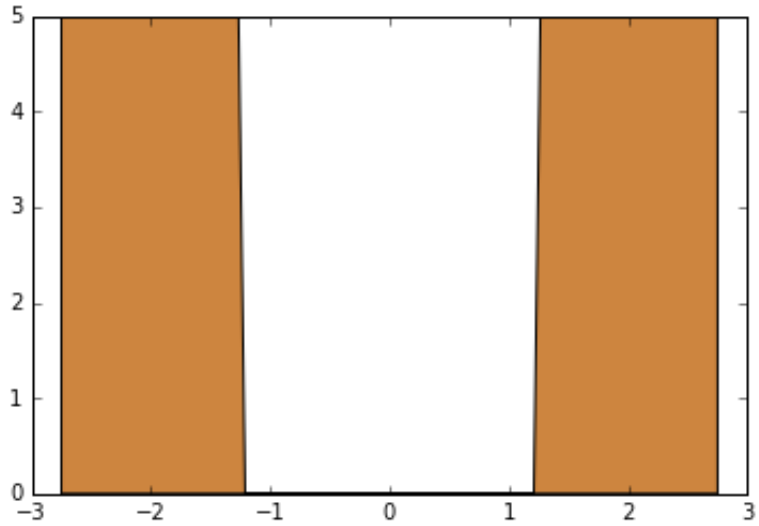
Para fines de ilustración y comprensión de los estados ligados del sistema, conviene poder ilustrar las funciones de potencial como barreras físicas. Esta noción gráfica se representa mediante el llenado entre la curva y el eje de referencia para la energía. De esta forma, al unir el gráfico con la referencia del autovalor, será claro que la energía hallada pertenece al intervalo requerido en teoría y que corresponde a un sistema ligado.

La función de graficación del potencial recibe dos listas/arreglos, uno con la información espacial y otro con la evaluación del potencial en dichos puntos. Antes de proceder con el llenado de la representación de la barrera del potencial, se crean los puntos inicial y final con el fin de crear formas cerradas distinguibles para el comando `fill`.

In [10]:
```
def graficar_potencial(x, V_x):
    V_min = min(V_x)
    plt.fill_between(x, V_min, V_x, facecolor = 'peru')
```

A continuación se presenta un ejemplo interactivo de graficación del potencial finito. Se inicia con la definición del potencial, la cual se usa para generar un arreglo con la información de la evaluación del potencial en distintos puntos del espacio.

In [11]:
```
def potencial(V_0, a, x):
    if abs(x) > a/2:
        return V_0
    else:
        return 0

def int_potencial(V_0:(.1, 10., .1), a:(.1, 5, .1), L:(1., 10.,
.5), N:(10, 200, 10)):
    dx = L / N
    x = [-L/2 + i*dx for i in range(N+1)]
    y = [potencial(V_0, a, i) for i in x]
    graficar_potencial(x, y)
    plt.show()

interact(int_potencial)
```



Out[11]: <function __main__.int_potencial>

Nivel de energía

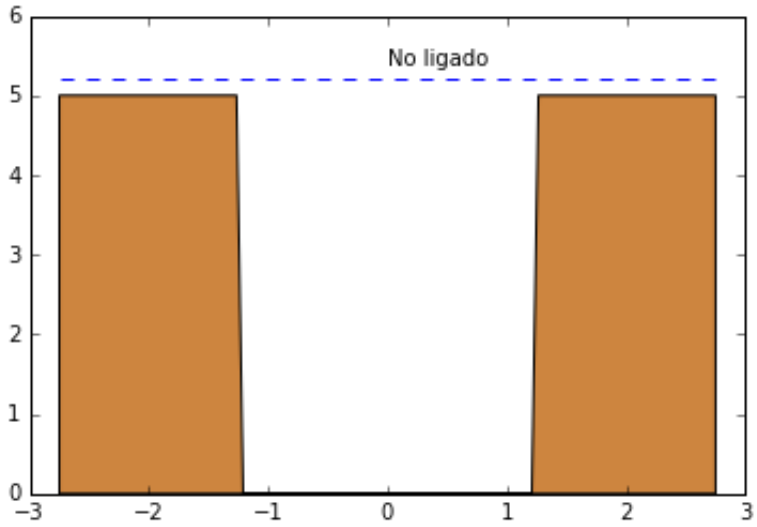
Para ilustrar adecuadamente la presencia de estados ligados, conviene superponer sobre la representación de la función de potencial, la referencia de energía del autovalor del sistema. Para distinguir este, sera un trazo discontinuo (no relleno para evitar confusión con el potencial, pero tampoco continuo para distinguirlo de la representación de las funciones de onda.

$$E \leq V_{máx}\,, \qquad \text{Estado ligado}$$
$$E > V_{máx}\,, \qquad \text{Estado no ligado}$$

Los estados no ligados son equivalentes a tener particulas libres.

In [12]:
```
def graficar_autovalor(L, E):
    plt.plot([-L/2, L/2], [E, E], '--')
```

In [13]:
```
def int_potencial_energia(V_0:(.1, 10., .1), E:(.1, 10., .1), a:
(.1, 5, .1), L:(1., 10., .5), N:(10, 200, 10)):
    dx = L / N
    x = [-L/2 + i*dx for i in range(N+1)]
    y = [potencial(V_0, a, i) for i in x]
    graficar_potencial(x, y)
    graficar_autovalor(L, E)
    if E > V_0:
        plt.text(0, E+0.2, 'No ligado')
    else:
        plt.text(0, E+0.2, 'Ligado')
    plt.show()

interact(int_potencial_energia)
```



Graficación de autofunciones

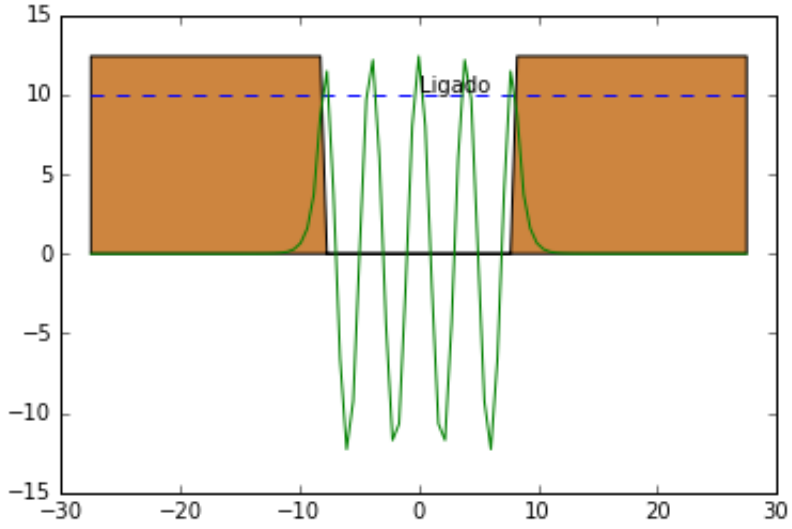
La visualización de las autofunciones (y su modulo cuadrado), nos permite visualmente reconocer la distribución de probabilidad del sistema e identificar los puntos espaciales más probables para la ubicación de la particula analizada.
Para la correcta visualización, la graficación de la función de onda debe considerar una normalización de escala, no necesariamente al valor de la unidad del eje, pero si como referencia un valor numerico comprendido por los valores máximos de potencial, que corresponden a la parte del gráfico más cercana al margen superior del recuadro de graficación. El no realizar este reescalamiento, podría afectar la visualización del potencial y de la energía, ya que el eje se reajusta a los datos maximos y minimos.

$$\psi'(x) = \frac{\psi(x)}{\max \psi(x)} V_{máx}$$

La graficación de las autofunciones es mediante el comando `plot` tradicional, y solo tiene de elemento adicional su reescalamiento con base al potencial máximo en la región de interes.

In [14]:
```
def graficar_autofuncion(x, psi_x, V_max):
    psi_max = max([abs(i) for i in psi_x])
    escala = V_max / psi_max
    psi_x = [i*escala for i in psi_x]
    plt.plot(x, psi_x, '-')
```

In [15]:
```
def onda(V_0, E, a, x):
    if abs(x) <= a/2:
        return cos(sqrt(E)*x/2)
    else:
        a2 = a/2
        k1 = sqrt(V_0 - E)
        A = cos(sqrt(E)*a2) / exp(-k1*a2)
        signo = abs(x)/x
        return A*exp(-signo*k1*x)

def int_potencial_auto_ef(V_0:(5., 20., .1), E:(.1, 20., .1), a:
(2.5, 30., .1), L:(10., 100., 5.), N:(10, 200, 10)):
    dx = L / N
    x = [-L/2 + i*dx for i in range(N+1)]
    V = [potencial(V_0, a, i) for i in x]
    f = [onda(V_0, E, a, i) for i in x]
    graficar_potencial(x, V)
    graficar_autovalor(L, E)
    graficar_autofuncion(x, f, V_0)
    if E > V_0:
        plt.text(0, E+0.2, 'No ligado')
    else:
        plt.text(0, E+0.2, 'Ligado')
    plt.show()

interact(int_potencial_auto_ef)
```



Out[15]: <function __main__.int_potencial_auto_ef>

Este notebook de ipython depende del modulo `vis_int`, el cual es ilustrado en el notebook de Visualización e Interacción (vis_int.ipybn).

In [1]:
```
from vis_int import *
import vis_int
print(dir(vis_int))
```

```
['Box', 'Button', 'FloatSlider', 'HTML', 'IntSlider', 'Latex', 'Ma
rkdown', 'Math', 'Text', '__builtins__', '__cached__', '__doc__',
'__file__', '__loader__', '__name__', '__package__', '__spec__',
'clear_output', 'cos', 'discretizar', 'display', 'exp', 'fixed',
'graficar_autofuncion', 'graficar_autovalor', 'graficar_funcion',
'graficar_potencial', 'graficar_punto_texto', 'interact', 'interac
tive', 'link', 'log', 'pi', 'plt', 'potencial', 'sin', 'sqrt', 'ta
n']
```

Técnicas Numéricas

Para el desarrollo de los modelos expuestos en los cursos de mecánica cuántica y física moderna, se recurre frecuentemente a funciones especiales y técnicas de solución matemáticas que en su operatividad pueden distraer el objetivo del curso y el adecuado entendimiento de los conceptos físicos en medio de las herramientas matemáticas.

Con esto en mente, el uso de técnicas numéricas simples puede apoyar significativamente el desarrollo de los cursos, teniendo como ventaja la reducción a formas matemáticas simples (los métodos numéricos llevan a aproximaciones con operaciones aritmeticas) y funciones simples (las funciones se aproximan a funciones mucho más simples, generalmente polinomios). Esta reducción facilita además reducir a una sola técnica multiples desarrollos, ya que las diferencias no van tanto en los detalles del modelo original (como si dependen las soluciones análiticas) sino en los detalles del tipo de aproximación general.

Se exponen las soluciones numéricas de los siguientes problemas, útiles para el desarrollo de problemas 1D de mecánica cuántica.

1. Búsqueda de raíces.
    - Bisección.
    - Incremental.
2. Ecuaciones diferenciales con valores de frontera.
    - Método del disparo con algoritmo Numerov.
3. Adimensionalización.
    - Unidades atomicas de Rydberg.

Búsqueda de raíces

Los problemas de búsquedas de raíces corresponden a encontrar valores que al evaluarse en la función de interes generan como evaluación el valor cero. En la mecánica cuántica nos encontramos con la particularidad de requerir el calculo de raíces para determinar los autovalores de energía de un sistema en su planteamiento continuo (representación en el espacio directo). En estos sistemas de interes, de estados ligados, la energía del sistema se encuentra entre el mínimo y el máximo de la energía potencial a la que se encuentra sometido en el espacio,

$$V_{mín} \leq E_n \leq V_{máx}.$$

En caso de ser el máximo $V_{máx} \rightarrow \infty$, el sistema posee infinitos autovalores que se encuentran con la condición $V_{mín} \leq E_n$.

Para cualquiera de los casos, se presenta un interes en encontrar estos autovalores de manera ordenada, y esto lleva seleccionar los métodos de búsqueda cerrados por encima de los métodos de búsquedas abiertos, ya que en estos últimos la selección de un valor inicial no asegura la búsqueda en cercanías de este o en una dirección dada, por el contrario en los métodos cerrados se puede limitar la búsqueda a una región de la cual tenemos conocimiento que se presenta la raíz (autovalor de energía).

El uso combinado entre el método de búsqueda incremental y el método de bisección (https://en.wikipedia.org/wiki/Bisection_method), con un paso adecuado de energía, permite cumplir con el objetivo de hallar todos los autovalores (cuando los límites de energía son finitos) del sistema de forma ordenada, y con precisión arbitraria (limitada solo por la precisión de máquina). Para ello se inicia en el intervalo de búsqueda con el método de búsqueda incremental, el cual al encontrar un intervalo candidato a raíz (un intervalo que presenta cambio de signo entre sus extremos), refina el resultado mediante la aplicación del método de bisección en el intervalo candidato.
Forma iterativa de búsqueda incremental $E_{i+1} = E_i + \Delta E$.
Forma iterativa de bisección $E_{i+1} = \frac{E_i + E_{i-1}}{2}$.

In [2]:
```
def biseccion(funcion, a, b, tol_x = 1e-6, factor_ty = 1e2):
    f0 = funcion(a)
    f1 = funcion(b)
    if abs(f0) < tol_x: # Se verifica que los extremos sean raices
        return a
    elif abs(f1) < tol_x:
        return b
    else: # Si los extremos no son raices, se bisecta.
        c = (a + b) / 2.0
        f2 = funcion(c)
        while abs(f2) >= tol_x and abs(c - b) >= tol_x:
            if f2 * f0 < 0 :
                b = c
                f1 = f2
            else:
                a = c
                f0 = f2
            c = (a + b) / 2.0
            f2 = funcion(c)
        if abs(f2) < tol_x * factor_ty: # Se verifica que efectivam
            return c
        else: # En caso de ser asintota vertical con cambio de sign
            return None

def incremental(funcion, a, b, delta_x = 1e-4, tol_x = 1e-6):
    c0 = a
    f0 = funcion(c0)
    c1 = c0 + delta_x
    c = None
    while c == None and c1 <=b: # Si no se ha hallado raíz y se est
        f1 = funcion(c1)
        while f0*f1 > 0 and c1 <= b:
            c0 = c1
            f0 = f1
            c1 = c1 + delta_x
            f1 = funcion(c1)
        if c1 > b: # Final del intervalo, equivalente f0*f1 > 0
            return None
        else: # Sub-intervalo con cambio de signo
            c = biseccion(funcion, c0, c1, tol_x) # Se invoca bisec
            if c == None: # Si el candidato era discontinuidad, inc
                c0 = c1
                f0 = f1
                c1 = c1 + delta_x
    return c
```

Se observa en la implementación del método de bisección, que se considera una revisión extra a los códigos tradicionales, con el fin de validad si el candidato a raíz realmente lo es. Esto se requiere ya que es posible que la función asociada a la discretización de la energía posea discontinuidades alrededor de las cuales presente cambio de signo. Notese que la teoría clásica de métodos numéricos indica que estos métodos se aplican

para funciones continuas. En este caso que esperamos discontinuidades dadas por cambios de signo a causa de divergencias al infinito, se pueden remover sistematicamente notando que a medida que se converge al candidato a raíz (tamaño de intervalo menor que la tolerancia), la evaluación de la función en este valor es significativamente mayor a la tolerancia, y cada vez su evaluación es mayor a la anterior.

$$E \in [E_i, Ei + 1] \wedge \Delta E \leq tol \wedge \begin{cases} f(E) > tol, & \text{Discontinuidad} \\ f(E) \leq tol, & \text{Raíz (autovalor)} \end{cases}$$

Una vez se obtiene una raíz, el método de búsqueda incremental continua nuevamente avanzando hasta encontrar un próximo intervalo candidato, al cual vuelve a aplicarle el método de bisección para distinguir si es raíz o discontinuidad. Este proceso se continua hasta el límite superior para la energía, $V_{máx}$.
Para la busqueda de un autovalor especifico, se requiere buscar todos los autovalores anteriores. De manera que se requiere de una función auxiliar que medie este progreso dada un modo. El caracter progresivo sobre las energías ofrece la ventaja sobre técnicas de autovalores, de la posibilidad de obtener los autovalores ordenados de manera natural.

In [3]:
```
def raiz_n(funcion, a, b, N, delta_x = 1e-4, tol_x = 1e-6):
    c0 = a
    cont_raiz = 0
    while c0 < b and cont_raiz < N:
        c = incremental(funcion, c0, b, delta_x, tol_x)
        if c == None: # Si incremental termina en 'None', no hay má
            return None
        cont_raiz = cont_raiz + 1
        c0 = c + delta_x
    if cont_raiz == N:
        return c
    else:
        return None
```

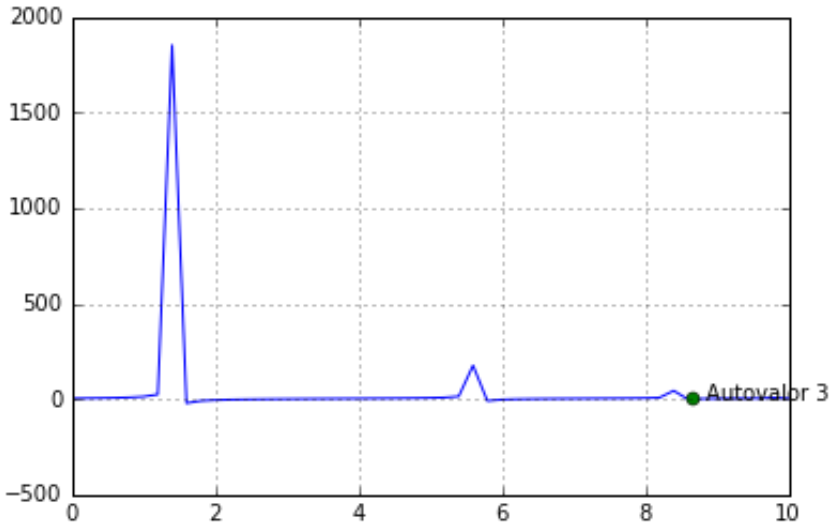
A continuación se ilustra el uso de la técnica con la función trascendental del problema del pozo finito simetrico con paridad par, que en la forma adimensional corresponde a:

$$\sqrt{E} - \sqrt{V_0 - E} \tan\left(\frac{\sqrt{V_0 - E}a}{2}\right) = 0,$$

con a el ancho del pozo, V_0 es la profundidad del pozo (con referencia desde cero por convención).

In [4]:
```
def trascendental(E, V_0, a):
    k2 = sqrt(V_0 - E)
    return sqrt(E) - k2*tan(k2*a/2)

def int_raiz_trasc(V_0:(.1,20.,.1), a:(.1,15.,.1), N:(1, 6, 1), n:(
    f = lambda E: trascendental(E, V_0, a)
    try:
        r = raiz_n(f, 0, V_0, N)
        E, tr = discretizar(f, 0, V_0, n)
        graficar_funcion(E, tr)
        graficar_punto_texto(r, 0, 'Autovalor ' + str(N))
        display(Latex('\(E_' + str(N) + '=' + str(r) + '\)'))
        plt.grid(True)
        plt.show()
        display(HTML('<div class="alert alert-warning">'+\
                     '<strong>Advertencia</strong> Alrededor de las
                     ' el gráfico no es representado fielmente. </d
    except ValueError:
        display(HTML('<div class="alert alert-danger">'+\
             '<strong>Error</strong> Se evaluo la función en una di
             '</div>'))

interact(int_raiz_trasc)
```

$E_3 = 8.661879687493155$



Advertencia Alrededor de las discontinuidades el gráfico no es representado fielmente.

Out[4]: <function __main__.int_raiz_trasc>

Ecuaciones diferenciales con problemas de frontera

La ecuación de Schrödinger, ya sea dependiente o independiente del tiempo, es una ecuación diferencial de segundo orden. Al remover la dependencia del tiempo y disponer de problemas 1D, se tiene que la ecuación diferencial es ordinaria. Para este tipo de ecuaciones (segundo orden, una variable) es posible aplicar métodos especificos que solución con bajo costo computacional aproximaciones de orden alto. Ejemplo de esto son los métodos de Verlet (https://en.wikipedia.org/wiki/Verlet_integration) y de Numerov (https://en.wikipedia.org/wiki/Numerov's_method), con método del disparo.

De la ecuación de Schrödinger se observa que si se reemplazan los valores por cantidades conocidas estimadas, el valor de la energía E que cumple con ser autovalor, es aquel que haga satisfacer las condiciones de frontera del problema, y por ende una forma de solucionar el problema es mediante la aplicación de un problema de busqueda de raices. De esta forma, el método del disparo lo que hace es el ajuste de E para que partiendo de una frontera, con la condicón respectiva, al propagarse hasta la otra frontera llegue con el valor de la otra condición. De no hacerlo, se cambio el valor de E y se repite el proceso.

El esquema de Numerov para la propagación es, dada una ecuación diferencial ordinaria de segundo orden sin termino lineal,

$$\frac{d^2y(x)}{dx^2} + K(x)y(x) = 0,$$

su esquema discreto se plantea como

$$y_{i+2} = \frac{\left(2 - \frac{5h^2K_{i+2}}{6}\right)y_{i+1} - \left(1 + \frac{h^2K_i}{12}\right)y_i}{\left(1 + \frac{h^2K_{i+2}}{12}\right)}.$$

Para nuestro caso, la función $K(x)$ posee dependencia de la energía, y todos los demás elementos son conocidos (la función solución, de onda en este caso, se construye iterativamente dado un valor de energía), por lo cual se puede definir una función que dada una energía como argumento, genere el valor de la función de onda en la frontera opuesta. Este valor en la frontera, por las condiciones establecidas por los potenciales y la condición de integrabilidad, debe ser $\psi(x_{izq}) = \psi(x_{der}) = 0$.

Tambien es usual usar como definición de la función, la diferencia de las derivadas logaritmicas en un punto intermedio, realizando la propagación desde ambos extremos. Por facilidad, se optará por la metodología clásica del disparo, que ofrece menor cantidad de operaciones y no presenta ambigüedades de definición, que consta de comparar con el valor esperado en la frontera opuesta, $Numerov(E) = 0$.

Este problema de búsqueda de raíces requiere conocer dos condiciones iniciales, la segunda debe tomarse acorde a la paridad de la energía buscada. Para paridad par, el segundo valor es positivo, mientras que para paridad impar el segundo valor es negativo.

La función `estacionario` se define para buscar el punto de empate adecuado para el análisis de la continuidad de la función de onda y su derivada. Como criterio, se buscan los *turning points* clásicos, donde $E = V(x)$.

In [5]:

```
def estacionario(K, L, h):
    x = -L/2
    while x < L/2 and K(x) <= 0:
        x = x + h
    if x >= L/2:
        return L/2
    elif x == -L/2:
        return -L/2
    else:
        return x - h

def numerov(K_ex, L, E, N, n):
    h = L / n
    K = lambda x: K_ex(E, x)
    p_est = estacionario(K, L, h)
    x = -L/2
    phi0 = 0.0
    x = x + h
    phi1 = 1e-10
    x = x + h
    while x <= p_est :
        term0 = 1 + h**2 * K(x - h) / 12
        term1 = 2 - 5 * h**2 * K( x) / 6
        term2 = 1 + h**2 * K(x + h) / 12
        aux = phi1
        phi1 = (term1 * phi1 - term0 * phi0) / term2
        phi0 = aux
        x = x + h
    phi_i_1 = phi1
    phi_i_0 = phi0
    x = L/2
    phi0 = 0.0
    x = x - h
    phi1 = 1e-10 * (-1)**(N%2 + 1)
    x = x - h
    while x > p_est :
        term0 = 1 + h**2 * K(x + h) / 12
        term1 = 2 - 5 * h**2 * K(x) / 6
        term2 = 1 + h**2 * K(x - h) / 12
        aux = phi1
        phi1 = (term1 * phi1 - term0 * phi0) / term2
        phi0 = aux
        x = x - h
    phi_d_1 = phi_i_1
    phi_d_0 = phi0 * phi_i_1 / phi1
    return (2*phi_d_1 - (phi_i_0+phi_d_0)) / (phi_d_0 - phi_i_0)


def Phi(K_ex, L, E, N, n):
    h = L / n
    K = lambda x: K_ex(E, x)
    p_est = estacionario(K, L, h)
    x = -L/2
    x_g = [x]
```

```
    x_g = [x]
    phi0 = 0.0
    phi_g = [phi0]
    x = x + h
    phi1 = 1e-10
    x_g.append(x)
    phi_g.append(phi1)
    x = x + h
    while x <= p_est:
        term0 = 1 + h**2 * K(x - h) / 12
        term1 = 2 - 5 * h**2 * K(x) / 6
        term2 = 1 + h**2 * K(x + h) / 12
        aux = phi1
        phi1 = (term1 * phi1 - term0 * phi0) / term2
        x_g.append(x)
        phi_g.append(phi1)
        phi0 = aux
        x = x + h
    x = L/2
    phi0 = 0.0
    x_gd = [x]
    phi_gd = [phi0]
    x = x - h
    phi1 = 1e-10 * (-1)**(N%2 + 1)
    x_gd.insert(0, x)
    phi_gd.insert(0, phi1)
    x = x - h
    while x > p_est:
        term0 = 1 + h**2 * K(x + h) / 12
        term1 = 2 - 5 * h**2 * K(x) / 6
        term2 = 1 + h**2 * K(x - h) / 12
        aux = phi1
        phi1 = (term1 * phi1 - term0 * phi0) / term2
        x_gd.insert(0, x)
        phi_gd.insert(0, phi1)
        phi0 = aux
        x = x - h
    n_d = len(phi_gd)
    phi_gd = [phi_gd[i] * phi_g[-1] / phi1 for i in range(n_d)]
    x_g.extend(x_gd)
    phi_g.extend(phi_gd)
    return x_g, phi_g
```

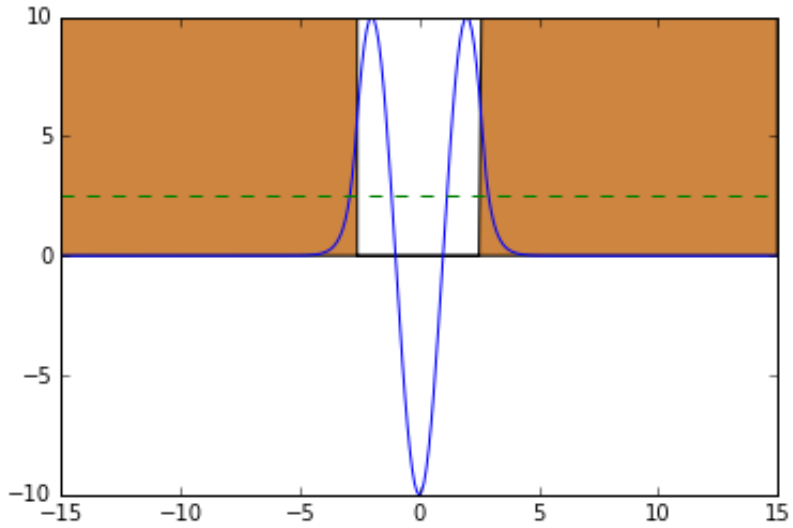
Para la ecuación de Schrödinger, $K(x) = E - V(x)$.

In [6]:
```
def K_Schr(V_0, a):
    return lambda e, x: e - potencial(V_0, a, x)
```

Para ilustrar el método del disparo, se presenta el siguiente control. La idea es ajustar para una configuración de potencial V_0, ancho a, longitud total L y numero de elementos de discretización n, la energía E adecuada para observar continuidad en la

función de onda y su derivada en todo el intervalo. Dada la implementación del método se verifica dicha continuidad en el limite de la primera pared. Más adelante se define de manera general como seleccionar el punto de comparación.

In [7]:
```
def disparo(V_0, a,  L, n, N, E):
    x, phi = Phi(K_Schr(V_0, a), L, E, N, n)
    V = [potencial(V_0, a, i) for i in x]
    graficar_potencial(x, V)
    graficar_autofuncion(x, phi, V_0)
    graficar_autovalor(L, E)
    plt.show()

def presion_disparo(boton):
    disparo(V_0, a.value,  L, n.value, N, E.value)

interact(disparo, V_0=(0., 20., .5), a=(.5, 10., .1), L=(10., 50.,
```



Out[7]: <function __main__.disparo>

La anterior ilustración tambien permite observar los efectos del potencial sobre un paquete de onda cuando la energía es menor o mayor que el potencial. Se puede observar como para $E > V_0$, se obtiene una función de onda oscilante en todo el intervalo, equivalente a una particula libre.

Se define la función `E_N` para el calculo de las autoenergías, que pueden incluirse en la función `Phi` para generar las autofunciones. Esta función puede explorarse en el notebook de Estados ligados (estados_ligados.ipynb).

In [8]:
```
def E_N(K, E_max, L, N, n, delta_e = 1e-4, tol_e = 1e-6):
    Numerov = lambda e: numerov(K, L, e, N, n)
    return raiz_n(Numerov, tol_e, E_max, N, delta_e, tol_e)
```

In [9]:
```
def Solve_Schr(Vx, E_max, L, N, n):
    x_vec, V_vec = discretizar(Vx, -L/2, L/2, n)
    V_min = min(V_vec)
    K = lambda e, x : e - Vx(x) + V_min
    E = E_N(K, E_max - V_min, L, N, n)
    if E != None:
        x_vec, phi = Phi(K, L, E, N, n)
        E = E + V_min
        display(Latex('\(E_{' + str(N) + '} = ' + str(E) + '\)'))
        V_vec = [Vx(i) for i in x_vec]
        graficar_potencial(x_vec, V_vec)
        V_max = max(V_vec)
        V_ref = max(abs(V_min), V_max)
        graficar_autofuncion(x_vec, phi, V_ref)
        graficar_autovalor(L, E)
        plt.show()
        return E, x_vec, phi
    else:
        display(HTML('<div class="alert alert-danger">'+\
             '<strong>Error</strong> Se evaluo la función en una di
             '</div>'))
```

El siguiente bloque define la base de los controles, `fun_contenedor_base`, para el notebook conceptual, Estados ligados (estados_ligados.ipynb), donde los parametros de máxima energía de búsqueda, longitud de interes, número de estado y particiones son comunes.

In [10]:
```
def fun_contenedor_base():
    E_max = FloatSlider(value=10., min = 1., max=20., step=1., desc
    L = FloatSlider(value = 30., min = 10., max = 100., step= 1., d
    N = IntSlider(value=1, min=1, max= 6, step=1, description='N')
    n = IntSlider(value= 300, min= 100, max= 500, step=20, descript
    return Box(children=[E_max, L, N, n])

Contenedor_base = fun_contenedor_base()
display(Contenedor_base)
```

In [11]:
```
def agregar_control(base, control):
    controles = list(base.children)
    controles.append(control)
    base.children = tuple(controles)

control_prueba = fun_contenedor_base()
agregar_control(control_prueba, Text(description='Casilla de texto
display(control_prueba)
```

La función `agregar_control` permite agregar controles adicionales a la base para crear el control especifico de los casos de prueba.

Adimensionalización

Para fines de la solución numérica, conviene definir las unidades atomicas de Rydberg (http://home.agh.edu.pl/~bjs/ARU.pdf). El uso de estas unidades permite hacer comparables los ordenes de maginitud tan dispares que poseen las variables involucradas y así controlar el error numérico que pueda tener el algoritmo.

$$\hbar = 2m_e = \frac{e^2}{2} = 1$$

Para fines de ilustración se considera que el problema se resuelve solo para electrones, de manera que el problema se hace independiente de la masa.

La energía se mide en Rydbergs y las longitudes en radios de Bohr.

$$1\ Ry = 13.6\ eV, \qquad 1\ a_0 = 5.29 \cdot 10^{-11}\ m$$

Realizando las sustituciones correspondienes, la forma adimensional de la ecuación Schrödinger es.

$$-\frac{d^2\psi(x)}{dx^2} + V(x)\psi(x) = E\psi(x)$$

Este notebook de ipython depende de los modulos:

- `tecnicas_numericas`, ilustrado en el notebook [Técnicas numéricas](tecnicas_numericas.ipynb).
- `vis_int`, ilustrado en el notebook [Visualización e Interacción](vis_int.ipybn) (esta incluido en el `import` a `tecnicas_numericas`).

In [1]:
```
from tecnicas_numericas import *
import tecnicas_numericas
print(dir(tecnicas_numericas))
```

```
['Box', 'Button', 'E_N', 'FloatSlider', 'HTML', 'IntSlider',
'K_Schr', 'Latex', 'Markdown', 'Math', 'Phi', 'Solve_Schr', 'Tex
t', '__builtins__', '__cached__', '__doc__', '__file__', '__load
er__', '__name__', '__package__', '__spec__', 'agregar_control',
'biseccion', 'clear_output', 'cos', 'discretizar', 'disparo', 'd
isplay', 'estacionario', 'exp', 'fixed', 'fun_contenedor_base',
'graficar_autofuncion', 'graficar_autovalor', 'graficar_funcio
n', 'graficar_potencial', 'graficar_punto_texto', 'incremental',
'interact', 'interactive', 'link', 'log', 'numerov', 'pi', 'pl
t', 'potencial', 'presion_disparo', 'raiz_n', 'sin', 'sqrt', 'ta
n']
```

Estados ligados

El presente documento cumple como función ilustrar de forma interactiva el comportamiento de las soluciones de la ecuación de Schrödinger estacionaria para estados ligados en problemas 1D con la aplicación del método de Numerov.

Simulación

La aplicación del método del disparo con el algoritmo de Numerov, implica la búsqueda de raíces para encontrar los autovalores de energía. Su forma de proceder es mediante el avance regular en pasos de energía entre un mínimo y máximo hasta encontrar un cambio de signo en la evaluación la función de onda (o criterio equivalente, como la derivada logaritmica de la misma) hasta el punto de comparación. La presencia de este cambio de signo indica que existe una energía E en el intervalo $[E_i, E_{i+1}]$ que es o raíz de la función de Numerov (por tanto autovalor del sistema) o una discontinuidad. Estas raíces y discontinuidas son asociadas a la función equivalente de cuantización de la energía, como en el problema típico de potencial finito lo es la ecuación trascendental (sin embargo, esta no aparace explicitamente en el modelo numerico).

Funciones de potencial

Las soluciones de autovalores de energía de un sistema se asocian al potencial y geometría del sistema (condiciones de frontera para un problema sobre una línea por ser 1D), las cuales se deben imponer acorde a las condiciones físicas de interes. Para las condiciones de frontera sabemos que en los extremos estas deben anularse, así que solo hace falta describir el potencial al cual se esta sujeto.

Pozo infinito y finito

El potencial del pozo infinito es descrito como:

$$V(x) = \begin{cases} 0 & |x| < \frac{a}{2} \\ \infty & |x| \geq \frac{a}{2} \end{cases}$$

Para el potencial de un pozo finito su potencial se puede describir como:

$$V(x) = \begin{cases} 0 & |x| < \frac{a}{2} \\ V_0 & |x| \geq \frac{a}{2} \end{cases}$$

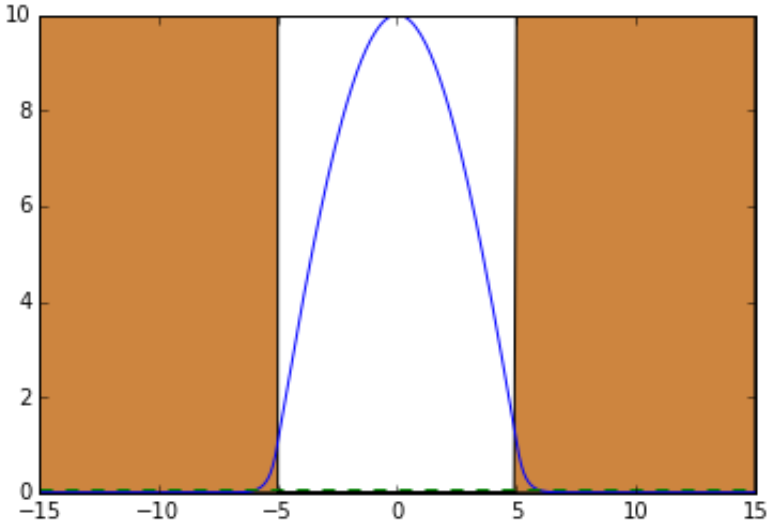
Estos pozos son los casos básicos de estudio por la facilidad para su desarrollo análitico e interpretación sencilla. Se puede ver en estos casos de estudio aplicaciones en ...

In [2]:
```
def V_inf(x):
    return 0

def V_fin(V_0, a, x):
    if abs(x) < a/2:
        return 0
    else:
        return V_0
```

Para efectos numéricos el infinito se traslada a una longitud grande comparativamente al ancho del pozo, la cual se designará como L. En el caso de que $a = L$, corresponde justamente al pozo infinito, de manera que la simulación de estos dos casos requiere un solo control y es basado en el potencial finito.

In [3]:
```
control_pozo = fun_contenedor_base()
agregar_control(control_pozo, FloatSlider(value = 5.2, min = .5,
max= 10., step= .1, description='a'))
pozo_link = link((control_pozo.children[1], 'min'), (control_poz
o.children[4], 'value'))
boton_pozo = Button(description='Simular pozo')
def click_pozo(boton):
    V_max = control_pozo.children[0].value
    L = control_pozo.children[1].value
    N = control_pozo.children[2].value
    n = control_pozo.children[3].value
    a = control_pozo.children[4].value
    Vx = lambda x: V_fin(V_max, a, x)
    Solve_Schr(Vx, V_max, L, N, n)
    clear_output(wait=True)

boton_pozo.on_click(click_pozo)
display(control_pozo, boton_pozo)
```

$E_1 = 0.08570021875000142$



Potencial armonico

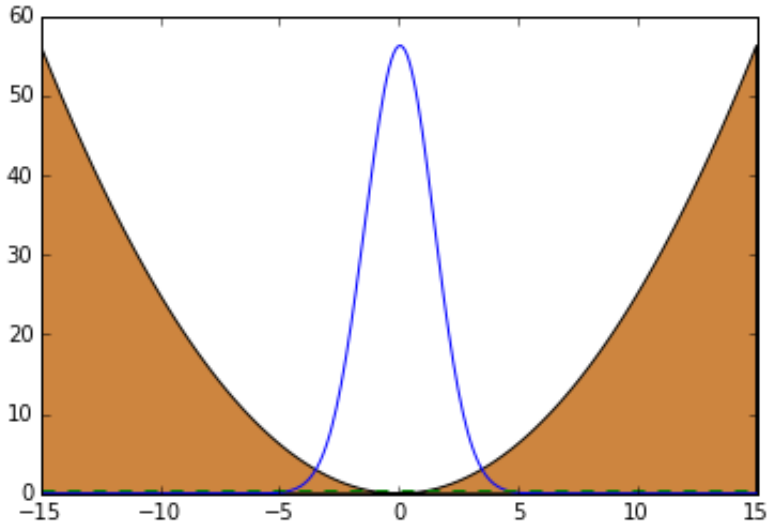
El potencial armonico cumple con la descripción dada por

$$V(x) = \frac{\omega^2 x^2}{4}$$

In [4]:
```
def V_arm(omega, x):
    return omega**2 * x**2 / 4
```

In [5]:
```
control_arm = fun_contenedor_base()
agregar_control(control_arm, FloatSlider(value = 1., min = .1, m
ax= 4., step= .1, description='$\omega$'))
boton_arm = Button(description='Simular potencial')
def click_arm(boton):
    E_max = control_arm.children[0].value
    L = control_arm.children[1].value
    N = control_arm.children[2].value
    n = control_arm.children[3].value
    omega = control_arm.children[4].value
    Vx = lambda x: V_arm(omega, x)
    Solve_Schr(Vx, E_max, L, N, n)
    clear_output(wait=True)

boton_arm.on_click(click_arm)
display(control_arm, boton_arm)
```

$E_1 = 0.48633928124996284$



Potencial arbitrario

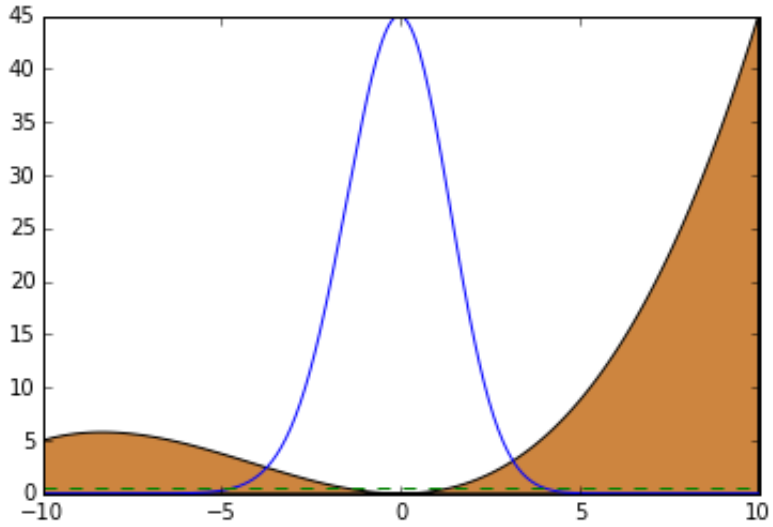
El problema 1D puede ser resuelto para un problema de potencial arbitrario $V(x)$, donde `str_potencial` es un *string* que representa la función del potencial. Debe tenerse en cuenta, tanto para este caso como los anteriores, que numéricamente el infinito es representado como una escala mayor por un cierta cantidad que la escala de interes del sistema.

Actividad : Proponga una función potencial de interes y desarrolle el bloque de código requerido para simularlo con este notebook. Use como base las funciones desarrolladas en los notebooks y el bloque siguiente.

El bloque siguiente ilustra un problema de potencial armonico con anarmonicidad.

In [6]:
```
control_arb = fun_contenedor_base()
E_max = control_arb.children[0]
L = control_arb.children[1]
N = control_arb.children[2]
n = control_arb.children[3]
n.value = 300
L.value = 20.

str_potencial = Text(value='x**2 / 4 + x**3 / 50', description=
'Potencial')
str_potencial.funcion = lambda x: eval(str_potencial.value)
agregar_control(control_arb, str_potencial)
# Ingrese un texto en formato python con dependencia solo de
'x'.

def ingreso_potencial(str_potencial):
    str_potencial.funcion = lambda x: eval(str_potencial.value)
    Vx = str_potencial.funcion
    h = L.value / n.value
    V_vec = [Vx(-L.value/2 + h*i) for i in range(n.value + 1)]
    V_min = min(V_vec)
    V_max = max(V_vec)
    dV = (V_max - V_min) / 50
    E_max.step = dV
    E_max.min = V_min
    E_max.max = V_max + (V_max - V_min)
    E_max.value = V_max

ingreso_potencial(str_potencial)
boton_arb = Button(description='Simular potencial')

def click_arbitrario(boton):
    Vx = str_potencial.funcion
    Solve_Schr(Vx, E_max.value, L.value, N.value, n.value)
    clear_output(wait=True)

str_potencial.on_submit(ingreso_potencial)
boton_arb.on_click(click_arbitrario)
display(control_arb, boton_arb)
```

$$E_1 = 0.4872908437499627$$



Para potenciales no simétricos y con amplias diferencias entre el potencial máximo y mínimo se recomienda usar instancias directas de python e ipython, debido al comportamiento como servidor de las instancias de Jupyter, que provocan la terminación del llamado al kernel tras un \textit{timeout}, de aproximadamente minuto y medio. Tras este periodo se recomienda reiniciar el kernel (tanto si es de ejecución local como en linea).

El caso con los siguientes parámetros toma 256s su solución y no es posible realizarlo en el notebook.

- $L = 20$
- $N = 1$
- $n = 300$
- $V(x) = \frac{x^2}{4} + 0.3x^3$
- $E_{max} = 325$

Se encuentra $E_1 = -241.0251$ y la función de onda graficada a continuación.