Select and Configure Theme

In [1]:

```python
#theme = 'grade3'   #classic
theme = 'oceans16' #dark mode
width = 1100 #px
```

Scroll Down for Content

In [2]:

```python
import warnings #tech debt
import math
import numpy as np
import matplotlib.pyplot as plt
import ipywidgets as widgets
```

In [3]:

```python
# styling
from IPython.display import HTML

#.css files were created using jupyter-themer and slightly modified:
# 1) text justification: added 2 lines to .rendered_html * + p block :
#      text-align: justify;
#      text-justify: inter-word;
# 2) fixing first paragraph of block:
#      created a copy of .rendered_html * + p block for a new
#        .rendered_html + p block (no asterisk!) and got rid of top margin
# 3) uniform line spacing of lines with and without inline latex:
#      increased lineheight from 130% to 200%
# 4) set bottom-margin of paragraphs to 1.5 em
#
# oceans16 specific:
# 1) added right-margin of 10 % for markdown text

#has to be last line of cell:
jt_style = open('./swap_invariants_'+theme+'.css').read()
jt_style = jt_style.replace('980px', str(width)+'px') #obviously hacky, but whatever
HTML('<style>'+jt_style+'</style>')
```

In [4]:

```python
#change style of plots
from jupyterthemes import jtplot
jtplot.style(theme=theme, context='notebook', figsize=(14,12))

#enable latex
fontsize = 22
plt.rc('text', usetex=True)
plt.rc('font', size=fontsize, family='serif')
plt.rc('axes', labelsize='x-large', titlesize='x-large')
plt.rc('xtick', labelsize=fontsize)
plt.rc('ytick', labelsize=fontsize)
plt.rc('legend', fontsize=fontsize)

default_colors = [cycle['color'] for _, cycle in enumerate(plt.rcParams["axes.prop_cycle"])]
#define uniform labels and colors for the different invariants for consistency between plots
def create_styles():
    styles = [
        ('sum', r'$I_\Sigma$', 0),
        ('prod', r'$I_\Pi$', 2),
        ('swim', '$I_C$', 5),
        ('chi', r'$I_\chi$', 4),
        ('kappa', r'$I_\kappa$', 9),
        ('theta', r'$I_\theta$', 10),
        ('stable_nn', r'$I_C$ ($A=A n^n$)', 11),
        ('curve', r'$I_C$ ($A=A n$, Curve)', 8),
    ]
    return dict([[name, {
        'label': label,
        'color': default_colors[color_index]
    }] for name, label, color_index in styles])
plot_styles = create_styles()
```

In [5]:

```python
eps = 0.00001
to_kappa = lambda lmbda: lmbda/(1-lmbda)

def I_sum_x_j(x_arr, D):
    return D - sum(x_arr)

def I_prod_x_j(x_arr, D):
    n = len(x_arr)+1
    return (D/n)**n / np.prod(np.maximum(x_arr, eps)) #np.minimum to prevent division by 0

def I_chi_x_j(x_arr, chi, D):
    n = len(x_arr)+1
    return (chi*(D - sum(x_arr)) + (D/n)**n)/(chi + np.prod(x_arr))

# haven't worked out x_j form for I_theta, but (r, phi) representation is good enough for plotting
def I_theta(theta, D, resolution):
    n = 2
    bound = np.pi/2-eps
    #use sin(linspace) to get higher resolution near 0 and pi/2 (where most of the action of I_theta is)
    phi = np.pi/4 * (1 + np.sin(np.linspace(-bound, bound, resolution)))
    r = theta*(D/(np.sin(phi)+np.cos(phi))) + (1-theta)*np.sqrt(2*D**n/(n**n * np.sin(2*phi)))
    return [r*np.cos(phi), r*np.sin(phi)] #return x and y array for plotting

def I_C_x_j_iterations(x_arr, A, D, rel_error = 1e-9, max_iterations = 100):
    n = len(x_arr) + 1
    x_arr = np.maximum(x_arr, eps)
    S = sum(x_arr)
    P = D/A * D/n
    for x_i in x_arr:
        P *= D/n/x_i

    x_j_approx = D #initial guess = average token balance
    for i in range(max_iterations):
        x_j_next = (x_j_approx ** 2 + P)/(2*x_j_approx + S + D/A - D)
        if 1 - rel_error < x_j_next/x_j_approx < 1 + rel_error:
            return (x_j_next, i)
        x_j_approx = x_j_next

    return (x_j_approx, -1)
```

In [6]:

```python
#n tokens - 2d visualization (trading x_i for x_j)
#mode in ['basics', 'theta', 'stableswap']
def visualize_invariants_2d(n, D, #basic parameters
                            zoom, equal_aspect, fixed_axis, resolution, #plot parameters
                            mode, lmbda, chi, with_theta, theta, A, rel_error): #mode parameters

    if mode != 'stableswap':
        assert(n==2)

    if mode == 'basics':
        kappa = to_kappa(lmbda)
        chi = kappa * D**(n-1)

    if theta == 0:
        theta = lmbda

    if A == 0:
        A = chi

    x_arr = [D/n] * (n-1)
    x_arr[0] = 0

    stop_y = fixed_axis if fixed_axis > 0 else I_chi_x_j(x_arr, chi, D)
    stop_x = D/n if zoom else stop_y

    #higher resolution closer to 0
    x = stop_x * np.linspace(0, 1, resolution)**2

    sum_x_j = np.zeros(resolution)
    chi_x_j = np.zeros(resolution)
    prod_x_j = np.zeros(resolution)
    if mode == 'basics':
        kappa_x_j = np.zeros(resolution)
    if mode == 'stableswap':
        C_x_j = np.zeros(resolution)
        C_iterations = np.zeros(resolution)

    for (k, x_i) in enumerate(x):
        x_arr[0] = x_i
        sum_x_j[k] = I_sum_x_j(x_arr, D)
        chi_x_j[k] = I_chi_x_j(x_arr, chi, D)
        prod_x_j[k] = I_prod_x_j(x_arr, D)
        if mode == 'basics':
            kappa_x_j[k] = I_chi_x_j(x_arr, kappa, D)
        if mode == 'stableswap':
            C_x_j[k], C_iterations[k] = I_C_x_j_iterations(x_arr, A, D, rel_error)

    fig, ax = plt.subplots()
    ax.plot(x, sum_x_j, **plot_styles['sum'])
    ax.plot(x, chi_x_j, **plot_styles['chi'])

    if mode == 'basics':
        ax.plot(x, kappa_x_j, **plot_styles['kappa'])

    if with_theta and n==2:
        theta_x_i, theta_x_j = I_theta(theta, D, resolution * (2 if zoom else 1))
        ax.plot(theta_x_i, theta_x_j, **plot_styles['theta'])

    if mode == 'stableswap':
        ax.plot(x, C_x_j, **plot_styles['swim'])

    ax.plot(x, prod_x_j, **plot_styles['prod'])

    to_dec_str = lambda f,digits: str(int(f*10**digits)/10**digits)
    title = 'Invariants\n$n='+str(n) + ', D='+to_dec_str(D,1)
    if mode == 'basics':
        title += r', \lambda='+to_dec_str(lmbda,2) + r', \kappa='+to_dec_str(kappa,1)
    title += r', \chi='+to_dec_str(chi,1)
    if mode == 'stableswap':
        title += ', A='+to_dec_str(A,1)
    if with_theta and n==2:
        title += r', \theta='+to_dec_str(theta,1)
    title += '$'
    plt.title(title)
    plt.xlabel('$x_i$')
    plt.ylabel('$x_j$')
    plt.grid(True)
    ax.legend()
    ax.set_xlim(left = 0, right = stop_x)
    ax.set_ylim(bottom = D/2 if zoom else 0, top = stop_y)
    if equal_aspect:
        ax.set_aspect('equal', adjustable='box')
    plt.show()

    if mode == 'stableswap':
        fig, ax = plt.subplots()
        plt.title('Iterations until Convergences of $x_j$ for $I_C$')
        plt.xlabel('$x_i$')
        plt.ylabel('$Iterations$')
        ax.plot(x, C_iterations)
        plt.show()

def interact_2d(mode):
    widgets.interact(visualize_invariants_2d,
        n            = widgets.IntSlider(min=2,max=6,step=1,value=2,continuous_update=False) if mode=='stablesw
        D            = widgets.FloatSlider(min=0.1,max=10,step=0.1,value=2,continuous_update=False),
        zoom         = widgets.Checkbox(value=False),
        equal_aspect = widgets.Checkbox(value=False),
        fixed_axis   = widgets.IntText(value=0,continuous_update=False),
        resolution   = widgets.IntText(value=50,continuous_update=False),
        mode         = widgets.fixed(mode),
        lmbda        = widgets.fixed(0) if mode!='basics' else widgets.FloatSlider(min=0.01,max=0.99,step=0.01
        chi          = widgets.FloatSlider(min=0.1,max=10,step=0.1,value=1,continuous_update=False) if mode!='
        with_theta   = widgets.Checkbox(value=True) if mode!='basics' else widgets.fixed(False),
        theta        = widgets.FloatSlider(min=0.01,max=0.99,step=0.01,value=0.5,continuous_update=False) if m
        A            = widgets.FloatLogSlider(base=10,min=-3,max=6,step=0.5,value=1,continuous_update=False)  i
        rel_error    = widgets.FloatLogSlider(base=10,min=-10,max=-1,step=0.5,value=-3,continuous_update=False
    )
```

In [7]:

```python
# 3 tokens 3d visualization
def visualize_3d(D, chi, zoom, fixed_axis, up_down, left_right, resolution):
    n = 3

    stop_z = fixed_axis if fixed_axis > 0 else I_chi_x_j([0,0], chi, D)
    stop_x = D/n if zoom else stop_z

    #higher resolution closer to 0
    x_1 = np.outer(stop_x * np.linspace(0, 1, resolution)**2, np.ones(resolution))
    x_2 = x_1.copy().T
    sum_x_3 = np.outer(np.zeros(resolution), np.zeros(resolution))
    prod_x_3 = np.outer(np.zeros(resolution), np.zeros(resolution))
    chi_x_3 = np.outer(np.zeros(resolution), np.zeros(resolution))

    for i in range(resolution):
        for j in range(resolution):
            x_arr = [x_1[i][j], x_2[i][j]]
            sum_x_3[i][j] = I_sum_x_j(x_arr, D)
            if sum_x_3[i][j] < 0 or sum_x_3[i][j] > stop_z:
                sum_x_3[i][j] = np.nan
            prod_x_3[i][j] = I_prod_x_j(x_arr, D)
            if prod_x_3[i][j] < 0 or prod_x_3[i][j] > stop_z:
                prod_x_3[i][j] = np.nan
            chi_x_3[i][j] = I_chi_x_j(x_arr, chi, D)
            if chi_x_3[i][j] < 0 or chi_x_3[i][j] > stop_z:
                chi_x_3[i][j] = np.nan

    with warnings.catch_warnings():
        warnings.filterwarnings('ignore') #technical debt of NaN cut-off handling
        fig = plt.figure()
        ax = plt.axes(projection ='3d')
        fig.set_size_inches(18,18)

        ax.plot_surface(x_1, x_2, sum_x_3, **plot_styles['sum'])
        ax.plot_surface(x_1, x_2, chi_x_3, **plot_styles['chi'])
        ax.plot_surface(x_1, x_2, prod_x_3, **plot_styles['prod'])

        ax.set(xlabel=r'$x_1$', ylabel=r'$x_2$', zlabel=r'$x_3$')
        ax.set_zlim(bottom = D/n if zoom else 0, top = stop_z)
        plt.title('Invariants\n'+r'$\chi='+str(int(chi*100)/100)+'$')

        ax.view_init(up_down, left_right)
        plt.show()

def interact_3d():
    widgets.interact(visualize_3d,
        D          = widgets.FloatSlider(min=0.1,max=10,step=0.1,value=3,continuous_update=False),
        chi        = widgets.FloatSlider(min=0.1,max=10,step=0.1,value=1,continuous_update=False),
        zoom       = widgets.Checkbox(value=False, description='zoom'),
        fixed_axis = widgets.IntText(value=2,continuous_update=False),
        up_down    = widgets.IntSlider(min=0,max=90,step=2,value=20,continuous_update=False),
        left_right = widgets.IntSlider(min=-90,max=90,step=2,value=0,continuous_update=False),
        resolution = widgets.IntText(value=50,continuous_update=False),
    )
```

In [8]:

```python
def amp_decay(n):
    resolution = 20
    x = np.linspace(1, resolution, resolution)
    #n different tokens, each with a balance of 1 in equilibrium, split into x pieces for nx pieces in total
    #first token has balance of 1 piece -> rho_1 = 1/x
    #n-1 pieces remaining split evenly over remaining n-1 tokens hence
    #rho_i = (nx-1)/(n-1)/x
    rho_1 = 1/x
    rho_i = (n*x - 1)/(x*(n-1))
    decay = rho_1 * rho_i**(n-1)

    fig, ax = plt.subplots()
    ax.bar(x, decay, label='Amp Decay')
    plt.xlabel(r'$\frac{1}{\rho}$')
    plt.ylabel('Amp Factor Scaling')
    plt.xticks(np.arange(1,resolution+1,1))
    plt.yticks(np.arange(0,1.1,0.1))
    plt.title('StableSwap Amp Decay (Lower Bound)')
    plt.grid(True)
    plt.show()

def interact_decay():
    widgets.interact(amp_decay,\
        n = widgets.IntSlider(min=2,max=6,step=1,value=2,continuous_update=False),\
    )
```

In [9]:

```python
def calculate_Curve_D(x_arr, A, rel_error = 1e-9, max_iterations = 100):
    n = len(x_arr)
    S = A*sum(x_arr)

    D_approx = sum(x_arr) #initial guess = D for pool in equilibrium
    for i in range(max_iterations):
        P = 1
        for x_i in x_arr:
            P *= D_approx/(n*x_i)

        D_next = (n*D_approx*P + S)/(A + (n+1)*P - 1)
        if 1 - rel_error < D_next/D_approx < 1 + rel_error:
            return (D_next, i)
        D_approx = D_next

    return (D_approx, -1)

def curve_D_2d(n, A, from_one, resolution, rel_error):
    x = np.linspace(0+eps,1-eps, resolution)
    D_cs = np.ones(resolution)
    D_C = np.zeros(resolution)
    D_C_n = np.zeros(resolution)
    D_C_nn = np.zeros(resolution)
    D_iterations = np.zeros(resolution)
    D_cp = np.zeros(resolution)
    x_arr = np.ones(n)/n
    for (i,x_i) in enumerate(x):
        if from_one:
            x_arr[0] = x_i/n
            x_arr[1] = (2-x_i)/n
        else:
            x_arr[0] = x_i/n
            for j in range(1,n):
                x_arr[j] = 1/n + (1 - x_i)/(n*(n-1))
        D_C[i], D_iterations[i] = calculate_Curve_D(x_arr, A, rel_error)
        D_C_n[i], _ = calculate_Curve_D(x_arr, A*n, rel_error)
        D_C_nn[i], _ = calculate_Curve_D(x_arr, A*(n**n), rel_error)
        D_cp[i] = n * pow(math.prod(x_arr),1/n)

    fig, ax = plt.subplots()
    if from_one:
        plt.title(r'$D$ as a function of $A, (\frac{x}{n}, \frac{2-x}{n}, \frac{1}{n}, \ldots (n-2) \textnormal
    else:
        plt.title(r'$D$ as a function of $A, (\frac{x}{n}, \frac{1}{n}+\frac{\frac{x}{n}}{n-1}, \ldots (n-1) \te
    plt.xlabel('$x$')
    plt.ylabel('$D$')
    ax.plot(x, D_cs, **plot_styles['sum'])
    ax.plot(x, D_C_nn, **plot_styles['stable_nn'])
    ax.plot(x, D_C_n, **plot_styles['curve'])
    ax.plot(x, D_C, **plot_styles['swim'])
    ax.plot(x, D_cp, **plot_styles['prod'])
    plt.legend()
    plt.show()

    fig, ax = plt.subplots()
    plt.title('Iterations until $I_C$ Convergence')
    plt.xlabel('$x$')
    plt.ylabel('Iterations')
    ax.plot(x, D_iterations, **plot_styles['swim'])
    plt.show()

def interact_D_2d():
    _ = widgets.interact(curve_D_2d,
        n          = widgets.IntSlider(min=2,max=6,step=1,value=2,continuous_update=False),
        A          = widgets.FloatLogSlider(base=10,min=-3,max=6,step=0.5,value=1,continuous_update=False),
        from_one   = widgets.Checkbox(value=True, description='take just from one (otherwise take from all equ
        resolution = widgets.IntText(value=200,continuous_update=False),
        rel_error  = widgets.FloatLogSlider(base=10,min=-10,max=-1,step=0.5,value=-3,continuous_update=False),
    )
```

In [10]:

```python
def visualize_D_3d(A, up_down, left_right, resolution, rel_error):
    x_1 = np.outer(np.linspace(eps, 1, resolution)**2, np.ones(resolution))
    x_2 = x_1.copy().T
    D = np.outer(np.zeros(resolution), np.zeros(resolution))

    for i in range(resolution):
        for j in range(resolution):
            D[i][j] = calculate_Curve_D([x_1[i][j], x_2[i][j]], A, rel_error)[0]

    fig = plt.figure()
    ax = plt.axes(projection ='3d')
    fig.set_size_inches(18,18)
    ax.plot_surface(x_1, x_2, D)
    ax.set(xlabel='$x_1$', ylabel='$x_2$', zlabel='$D$')
    plt.title('D as a function of $A, (x_1, x_2)$\n$A='+str(int(A*100)/100)+'$')
    ax.view_init(up_down, left_right)
    plt.show()

def interact_D_3d():
    widgets.interact(visualize_D_3d,
        A          = widgets.FloatLogSlider(base=10,min=-3,max=6,step=0.5,value=1,continuous_update=False),
        up_down    = widgets.IntSlider(min=0,max=90,step=2,value=20,continuous_update=False),
        left_right = widgets.IntSlider(min=-180,max=180,step=2,value=-124,continuous_update=False),
        resolution = widgets.IntText(value=50,continuous_update=False),
        rel_error  = widgets.FloatLogSlider(base=10,min=-10,max=-1,step=0.5,value=-3,continuous_update=False),
    )
```

In [11]:

```python
def visualize_slippage(n, A, upper, resolution, rel_error, amp_corr):
    x = np.linspace(eps, upper, resolution)
    x_j = [np.zeros(resolution) for _ in range(3)]
    slippage = [np.zeros(resolution) for _ in range(3)]

    amp = A*n if amp_corr else A

    for i in range(resolution):
        x_arr = [1]*(n-2) + [1+x[i]]
        x_j[0][i] = I_sum_x_j(x_arr, n) if x[i] <= 1 else float('nan')
        x_j[1][i] = I_C_x_j_iterations(x_arr, amp, n, rel_error)[0]
        x_j[2][i] = I_prod_x_j(x_arr, n)
        for k in range(3):
            slippage[k][i] = (1 - (1-x_j[k][i])/x[i]) * 100

    fig, ax = plt.subplots()
    plt.title('Equilibrium Slippage')
    plt.xlabel('token balance added (\%)')
    plt.ylabel('slippage (\%)')
    ax.plot(x*100, slippage[0], **plot_styles['sum'])
    ax.plot(x*100, slippage[1], **plot_styles['swim'])
    ax.plot(x*100, slippage[2], **plot_styles['prod'])
    ax.legend()
    plt.show()

    fig, ax = plt.subplots()
    plt.title('Amount')
    plt.xlabel('added')
    plt.ylabel("received")
    ax.plot(x, 1-x_j[0], **plot_styles['sum'])
    ax.plot(x, 1-x_j[1], **plot_styles['swim'])
    ax.plot(x, 1-x_j[2], **plot_styles['prod'])
    ax.legend()
    ax.plot()
    plt.show()

def interact_slippage():
    widgets.interact(visualize_slippage,
        n          = widgets.IntSlider(min=2,max=6,step=1,value=2,continuous_update=False),
        A          = widgets.FloatLogSlider(base=10,min=-3,max=6,step=0.5,value=1,continuous_update=False),
        upper      = widgets.FloatSlider(min=0.01,max=5,step=0.01,value=1,continuous_update=False),
        resolution = widgets.IntText(value=200,continuous_update=False),
        rel_error  = widgets.FloatLogSlider(base=10,min=-10,max=-1,step=0.5,value=-3,continuous_update=False),
        amp_corr   = widgets.Checkbox(value=False, description='multiply A by n for proper scale invariance'),
    )
```

In [12]:

```python
from swim_invariant import SwimPool, Decimal

def marginal_prices_outer(n):
    minstep = 0.05
    sliders = [widgets.FloatSlider(
        min=minstep,
        max=n-((n-1)*minstep)+eps,
        step=minstep,
        value=1,
        orientation='vertical',
        continuous_update=False,
        layout=widgets.Layout(width='50px')
    ) for i in range(n)]

    locks = [widgets.Checkbox(
        value=False,
        description='🔒',
        indent=False,
        layout=widgets.Layout(width='50px')
    ) for i in range(n)]

    scale = widgets.Dropdown(
        options=[('D', n)] + [("{}".format(i+1),i) for i in range(n)],
        value=n,
        description='relative to',
    )
    amp = widgets.FloatLogSlider(description="A",base=10,min=-3,max=6,step=0.5,value=1,continuous_update=False)

    updating = False
    def slider_moved(args):
        nonlocal updating
        if updating:
            return

        updating = True

        index = sliders.index(args.owner)
        vals = [slider.value for slider in sliders]

        diff = n-sum(vals) #positive if other balances need to be scaled up, negative otherwise
        unlocked_indices = [i for i in range(n) if (not locks[i].value and i != index)]
        while len(unlocked_indices) > 0:
            cur_index = unlocked_indices[0]
            x = diff/sum([vals[i] for i in unlocked_indices])
            shift = vals[cur_index]*x
            diff -= shift
            unclamped = vals[cur_index]+shift
            if unclamped > n:
                diff += unclamped-n
                vals[cur_index] = n
            elif unclamped < minstep:
                diff += unclamped-minstep
                vals[cur_index] = minstep
            else:
                vals[cur_index] = unclamped
            unlocked_indices = unlocked_indices[1:]
        vals[index] += diff

        for i in range(n):
            sliders[i].value = vals[i]
        updating = False

    for slider in sliders:
        slider.observe(slider_moved, 'value')

    def marginal_prices_inner(**args):
        amp = args['amp']
        scale = args['scale']
        balances = [args['balance{}'.format(i)] for i in range(n)]
        pool = SwimPool(n, Decimal(amp), 0 , 0, eps)
        pool.add(balances)
        mp = pool.marginal_prices()
        if scale < n:
            mp = [price/mp[scale] for price in mp]
        mp = [round(price, 2) for price in mp]
        D = pool.depth()
        x = np.arange(1,n+1)
        fig, ax = plt.subplots()

        b = ax.bar(x, mp)
        ax.bar_label(b)
        plt.xlabel('balances')
        plt.ylabel('prices')
        plt.xticks(np.arange(1,n+1))
        #plt.yticks(np.arange(0.5,1.0,0.1))
        plt.title('Marginal Prices - $D='+str(int(D*100)/100)+'$')
        plt.grid(True)
        plt.show()

    slider_vboxes = [widgets.VBox([sliders[i], locks[i]]) for i in range(n)]
    master_box = widgets.VBox([widgets.HBox([amp, scale]), widgets.HBox(slider_vboxes)])

    inner_args = {'balance{}'.format(i): sliders[i] for i in range(n)}
    inner_args['amp'] = amp
    inner_args['scale'] = scale
    out = widgets.interactive_output(marginal_prices_inner, inner_args)
    display(master_box, out)

def interact_marginal_prices():
    widgets.interact(marginal_prices_outer, n=widgets.IntSlider(min=2, max=6, step=1, value=2))
```

Purpose of this Notebook

This notebook explains/documents the (deriviation of) the math used in Swim's pool contract, i.e. how swaps, adds, removes, and their associated fees are calculated. It does not deal with any issues that are specific to implementing this math on the Solana blockchain and it's (at the time of writing) rather limited datatypes for this sort of math. The goal of this notebook is to illuminate the mathematics and rationales that gave rise to the pool contract's code.

The first part of the notebook deals with pool operations in the absence of fees, which are subsequently included in the second part.

Basic Invariants

In order to implement an AMM (automated market maker), a formula is required that defines the exchange rate between the tokens of the liquidity pool at different liquidity levels (= token balances of the pool). Essentially, such a formula defines what liquidity levels are to be considered equivalent, which then in turn defines how many tokens of some type a user has to transfer to the pool in order to receive a certain amount of tokens of some other type (we call this operation a swap).

More formally, assume a pool with n different tokens with balances $x_i, i = 1, \ldots, n$ (we're mostly sticking to the notation of the original curve whitepaper).

First, consider the constant **sum** invariant (where D can be considered the depth of the pool):

$$I_\Sigma := \left[\sum_{i=1}^n x_i = D\right]$$

(The constant sum invariant essentially says: All tokens are always equally valuable, regardless of their relative abundance within the liquidity pool. Hence each token can always be swapped at a 1:1 ratio with any other token in the pool. So a pool can e.g. be taken from $(1, 1)$ to $(0.5, 1.5)$, i.e. one needs to send 0.5 of the second token to receive 0.5 of the first.)

If we assume that there is an equal number of all tokens in the pool, i.e. $\forall i, j : x_i = x_j$ (we also call such a pool in equilibrium) then $x_i = \frac{D}{n}$ and hence if we were to apply the constant **product** invariant instead, we'd get:

$$I_\Pi := \left[\prod_{i=1}^n x_i = \left(\frac{D}{n}\right)^n\right]$$

(So instead of keeping the sum of the token balances invariant under swaps and hence treating every token as if they are equally valuable at all times, the constant product invariant holds the product constant thus to receive half the balance of one type of token in the pool, you have to send double the amount of the other token, i.e. to take 0.5 of the first token for a pool with balances $(1, 1)$, you actually have to send 1 full token of second type (unlike 0.5 as for the constant sum).)

Now that we have two invariants I_Σ and I_Π, any convex combination of the two will yield a new invariant that lies between the two:

$$I_\lambda := \lambda I_\Sigma + (1 - \lambda) I_\Pi \quad, \lambda \in (0, 1)$$

Alternatively, we can divide both sides of I_λ by $1 - \lambda$ and define $\kappa = \frac{\lambda}{1-\lambda}$ to get an alternative form:

$$I_\kappa := \kappa I_\Sigma + I_\Pi \quad, \kappa \in (0, \infty)$$

However, choosing κ (or λ) independent of D is suboptimal because it leads to our interpolation not being scale-free: If we consider a pool with n tokens that's maximally out of balance, i.e. $\forall i > 1 : x_i = 0$, we get:

$$x_1 = D + \frac{1}{\kappa}\left(\frac{D}{n}\right)^n$$

which is not linear in D.

Thankfully, this problem is easily rectified by choosing $\chi := \kappa D^{n-1}$, giving rise to our scale-free invariant I_χ:

$$I_\chi := \chi I_\Sigma + I_\Pi$$

Plugging in, we get:

$$I_\chi = \left[\chi \sum_{i=1}^n x_i + \prod_{i=1}^n x_i = \chi D + \left(\frac{D}{n}\right)^n\right]$$

Visualization

To get an idea what these invariants actually look like, let's plot them for a pool with $n = 2$ and $n = 3$ tokens respectively.

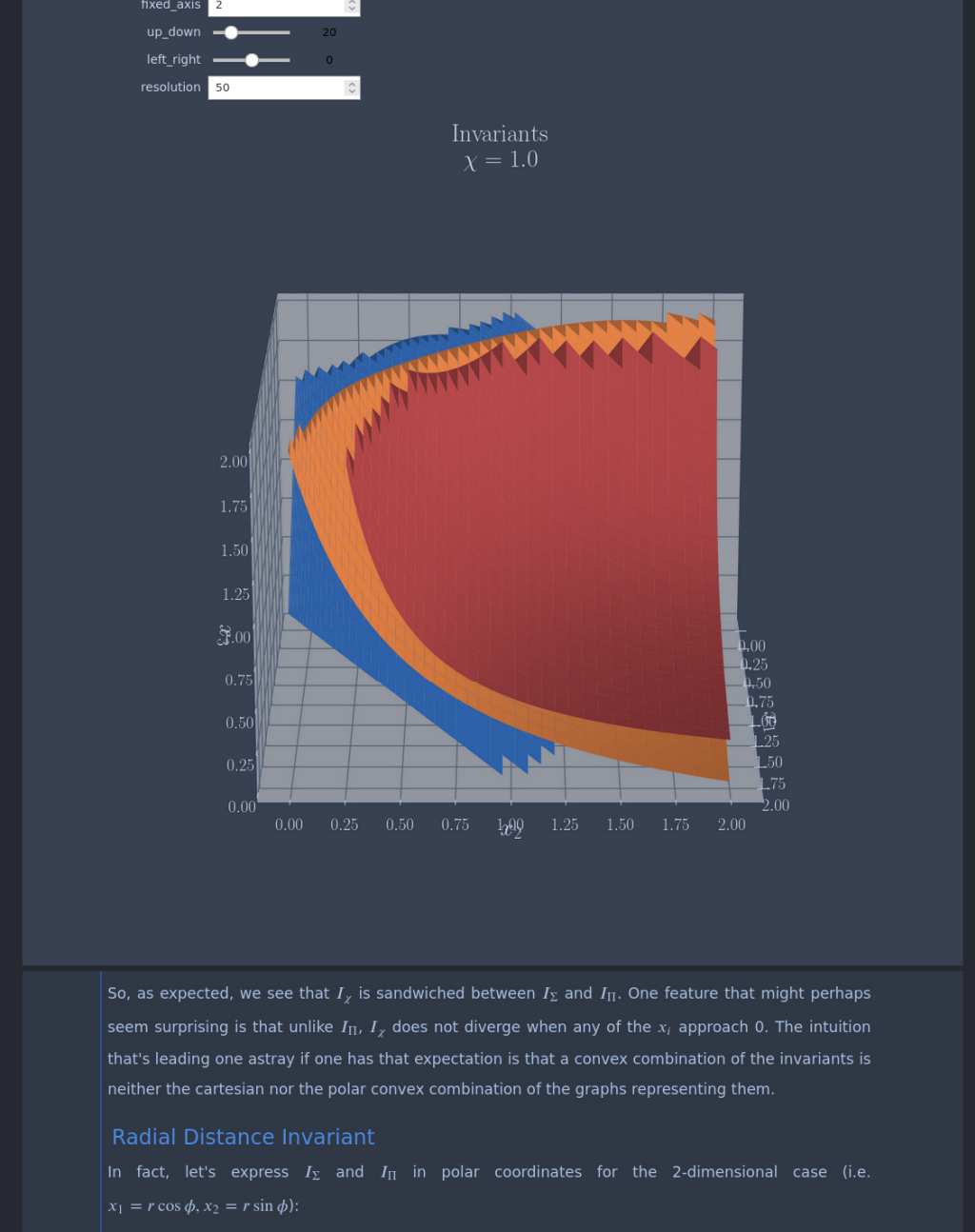
To that end, we need to express x_j as a function of all other $x_i, i \neq j$, D, and χ. With a bit of rearranging we get:

$$x_j = \frac{\chi\left(D - \sum_{i \neq j} x_i\right) + \left(\frac{D}{n}\right)^n}{\chi + \prod_{i \neq j} x_i}$$

Try changing D and see how I_κ moves around while the other 3 invariants stay in place:

In [13]:

```python
interact_2d('basics')
```



D 2.00

☐ zoom

☐ equal_aspect

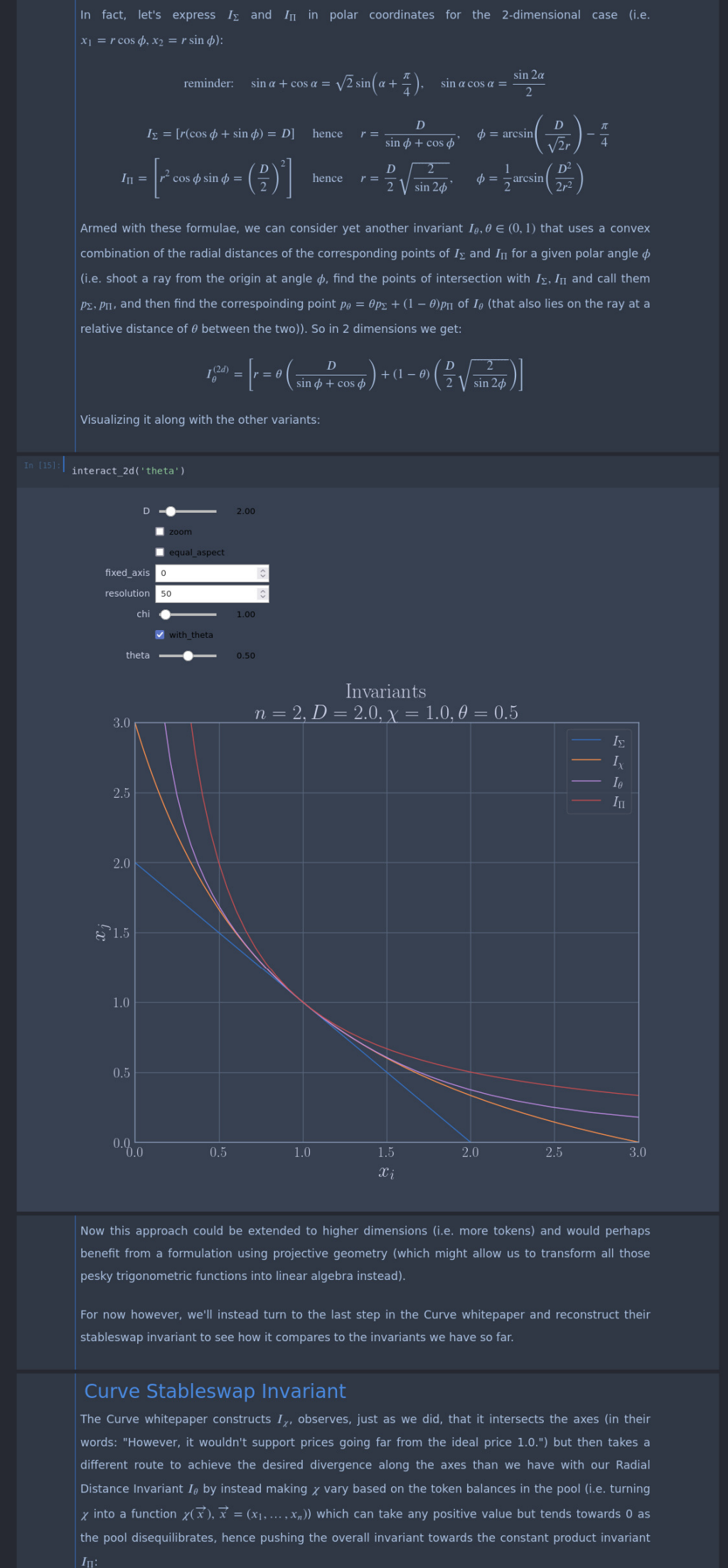
fixed_axis 0

resolution 50

lmbda 0.50

In [14]:

```python
interact_3d()
```



D 3.00

chi 1.00

☐ zoom

fixed_axis 2

up_down 20

left_right 0

resolution 50

So, as expected, we see that I_χ is sandwiched between I_Σ and I_Π. One feature that might perhaps seem surprising is that unlike I_Π, I_χ does not diverge when any of the x_i approach 0. The intuition that's leading one astray if one has that expectation is that a convex combination of the invariants is neither the cartesian nor the polar convex combination of the graphs representing them.

Radial Distance Invariant

In fact, let's express I_Σ and I_Π in polar coordinates for the 2-dimensional case (i.e. $x_1 = r\cos\phi, x_2 = r\sin\phi$):

$$\text{reminder:} \quad \sin\alpha + \cos\alpha = \sqrt{2}\sin\left(\alpha + \frac{\pi}{4}\right), \quad \sin\alpha\cos\alpha = \frac{\sin 2\alpha}{2}$$

$$I_\Sigma = [r(\cos\phi + \sin\phi) = D] \quad \text{hence} \quad r = \frac{D}{\sin\phi + \cos\phi}, \quad \phi = \arcsin\left(\frac{D}{\sqrt{2}r}\right) - \frac{\pi}{4}$$

$$I_\Pi = \left[r^2\cos\phi\sin\phi = \left(\frac{D}{2}\right)^2\right] \quad \text{hence} \quad r = \frac{D}{2}\sqrt{\frac{2}{\sin 2\phi}}, \quad \phi = \frac{1}{2}\arcsin\left(\frac{D^2}{2r^2}\right)$$

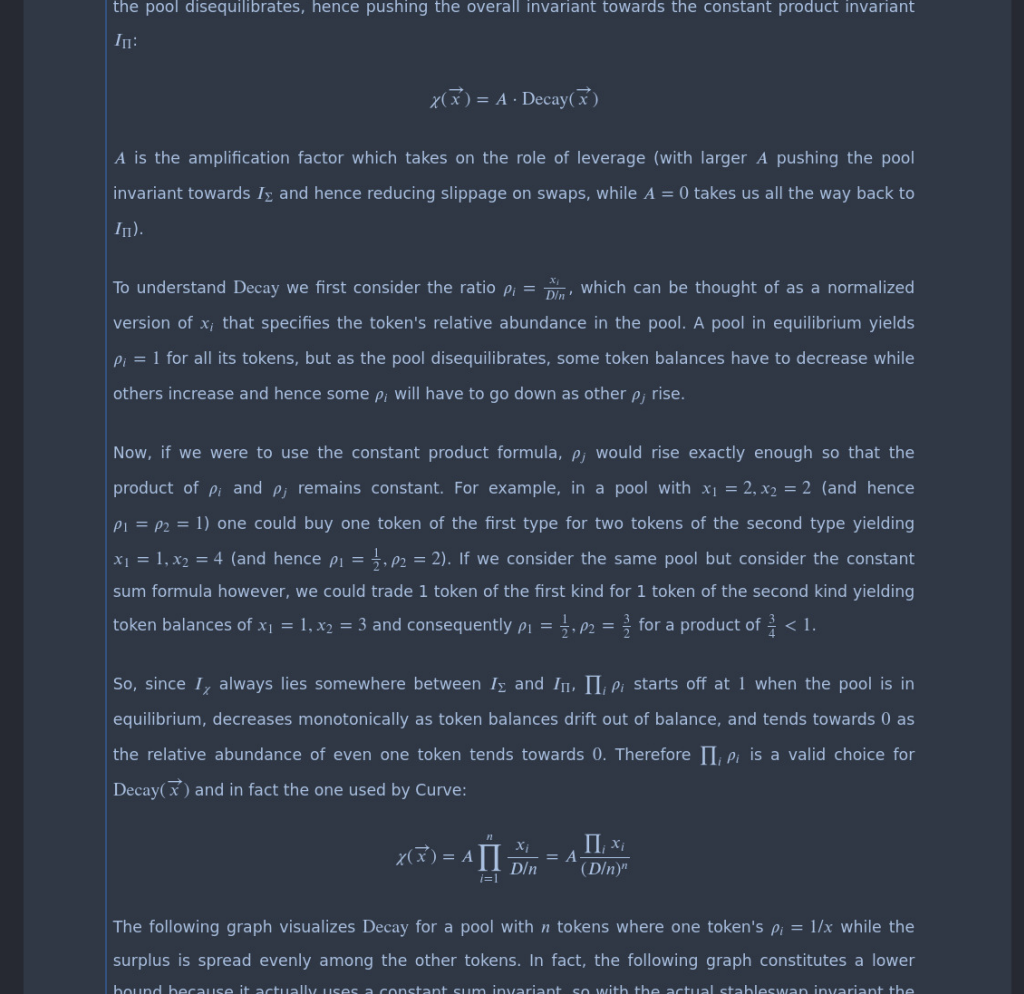
Armed with these formulae, we can consider yet another invariant $I_\theta, \theta \in (0, 1)$ that uses a convex combination of the radial distances of the corresponding points of I_Σ and I_Π for a given polar angle ϕ (i.e. shoot a ray from the origin at angle ϕ, find the points of intersection with I_Σ, I_Π and call them p_Σ, p_Π, and then find the correspoinding point $p_\theta = \theta p_\Sigma + (1 - \theta) p_\Pi$ of I_θ (that also lies on the ray at a relative distance of θ between the two)). So in 2 dimensions we get:

$$I_\theta^{(2d)} = \left[r = \theta\left(\frac{D}{\sin\phi + \cos\phi}\right) + (1 - \theta)\left(\frac{D}{2}\sqrt{\frac{2}{\sin 2\phi}}\right)\right]$$

Visualizing it along with the other variants:

In [15]:

```python
interact_2d('theta')
```



D 2.00

☐ zoom

☐ equal_aspect

fixed_axis 0

resolution 50

chi 1.00

☑ with_theta

theta 0.50

Now this approach could be extended to higher dimensions (i.e. more tokens) and would perhaps benefit from a formulation using projective geometry (which might allow us to transform all those pesky trigonometric functions into linear algebra instead).

For now however, we'll instead turn to the last step in the Curve whitepaper and reconstruct their stableswap invariant to see how it compares to the invariants we have so far.

Curve Stableswap Invariant

The Curve whitepaper constructs I_χ, observes, just as we did, that it intersects the axes (in their words: "However, it wouldn't support prices going far from the ideal price 1.0.") but then takes a different route to achieve the desired divergence along the axes than we have with our Radial Distance Invariant I_θ by instead making χ vary based on the token balances in the pool (i.e. turning χ into a function $\chi(\vec{x}), \vec{x} = (x_1, \ldots, x_n)$) which can take any positive value but tends towards 0 as the pool disequilibrates, hence pushing the overall invariant towards the constant product invariant I_Π:

$$\chi(\vec{x}) = A \cdot \text{Decay}(\vec{x})$$

A is the amplification factor which takes on the role of leverage (with larger A pushing the pool invariant towards I_Σ and hence reducing slippage on swaps, while $A = 0$ takes us all the way back to I_Π).

To understand Decay we first consider the ratio $\rho_i = \frac{x_i}{D/n}$, which can be thought of as a normalized version of x_i that specifies the token's relative abundance in the pool. A pool in equilibrium yields $\rho_i = 1$ for all its tokens, but as the pool disequilibrates, some token balances have to decrease while others increase and hence some ρ_i will have to go down as other ρ_j rise.

Now, if we were to use the constant product formula, ρ_j would rise exactly enough so that the product of ρ_i and ρ_j remains constant. For example, in a pool with $x_1 = 2, x_2 = 2$ (and hence $\rho_1 = \rho_2 = 1$) one could buy one token of the first type for two tokens of the second type yielding $x_1 = 1, x_2 = 4$ (and hence $\rho_1 = \frac{1}{2}, \rho_2 = 2$). If we consider the same pool but consider the constant sum formula however, we could trade 1 token of the first kind for 1 token of the second kind yielding token balances of $x_1 = 1, x_2 = 3$ and consequently $\rho_1 = \frac{1}{2}, \rho_2 = \frac{3}{2}$ for a product of $\frac{3}{4} < 1$.

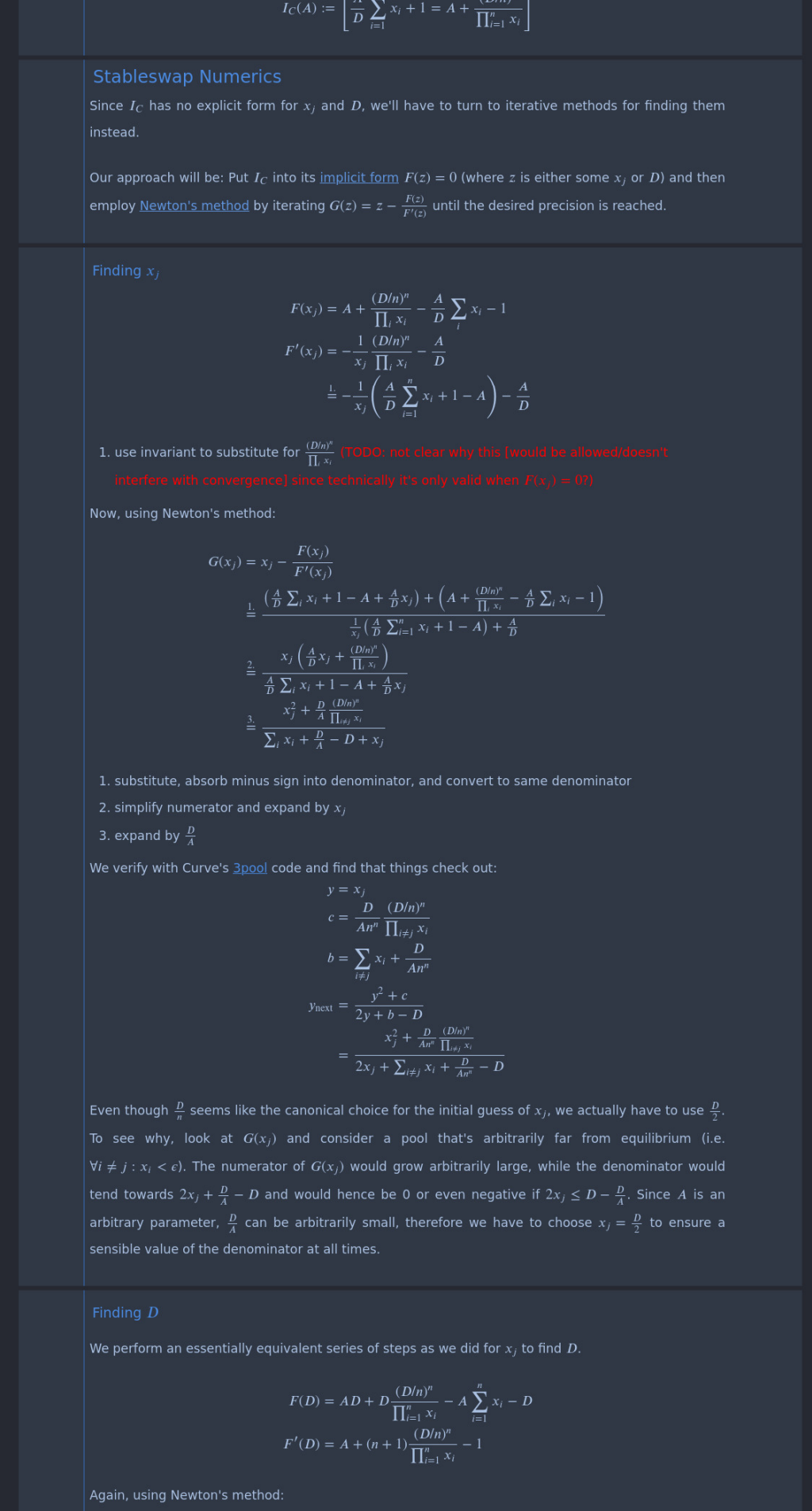
So, since I_χ always lies somewhere between I_Σ and I_Π, $\prod_i \rho_i$ starts off at 1 when the pool is in equilibrium, decreases monotonically as token balances drift out of balance, and tends towards 0 as the relative abundance of even one token tends towards 0. Therefore $\prod_i \rho_i$ is a valid choice for $\text{Decay}(\vec{x})$ and in fact the one used by Curve:

$$\chi(\vec{x}) = A\prod_{i=1}^n \frac{x_i}{D/n} = A\frac{\prod_i x_i}{(D/n)^n}$$

The following graph visualizes Decay for a pool with n tokens where one token's $\rho_i = 1/x$ while the surplus is spread evenly among the other tokens. In fact, the following graph constitutes a lower bound because it actually uses a constant sum invariant, so with the actual stableswap invariant the decay would be even slower (i.e. longer bars):

In [16]:

```python
interact_decay()
```



n 2

Notice that there are plenty of other, readily available candidates for $Decay$ such as $\min_i(\rho_i)$ and that if f is a valid choice for $Decay$, so is $f^s, s > 0$.

Plugging $\chi(\vec{x})$ into I_χ and dividing by $\prod_i x_i$ we get:

$$\frac{An^n}{D}\sum_{i=1}^n x_i + 1 = An^n + \underbrace{\frac{(D/n)^n}{\prod_{i=1}^n x_i}}_{=\text{Decay}^{-1}}$$

We observe that A always comes with the factor of n^n. Absorbing it into A has two advantages: first, it reduces clutter and secondly, and more importantly, it actually makes the overall shape of the invariant more comparable between different values of n ([illegible] further below).

So with this final modification we arrive at the CurveStableswap invariant I_C:

$$I_C := \left[A\sum_i x_i + D = AD + D\frac{(D/n)^n}{\prod_i x_i}\right]$$

Stableswap Numerics

Since I_C has no explicit form for x_j and D, we'll have to turn to iterative methods for finding them instead.

Our approach will be: Put I_C into its implicit form $F(z) = 0$ (where z is either some x_j or D) and then employ Newton's method by iterating $G(z) = z - \frac{F(z)}{F'(z)}$ until the desired precision is reached.

Finding x_j

$$\begin{aligned} F(x_j) &= A + \frac{(D/n)^n}{\prod_i x_i} - \frac{A}{D}\sum_i x_i - 1 \\ F'(x_j) &= -\frac{1}{x_j}\frac{(D/n)^n}{\prod_i x_i} - \frac{A}{D} \\ &\overset{1.}{=} -\frac{1}{x_j}\left(\frac{A}{D}\sum_{i=1}^n x_i + 1 - A\right) - \frac{A}{D} \end{aligned}$$

1. use invariant to substitute for $\frac{(D/n)^n}{\prod_i x_i}$ (TODO: not clear why this (would be allowed/doesn't interfere with convergence) since technically it's only valid when $F(x_j) = 0$!)

Now, using Newton's method:

$$\begin{aligned} G(x_j) &= x_j - \frac{F(x_j)}{F'(x_j)} \\ &\overset{1.}{=} \frac{\left(\frac{A}{D}\sum_i x_i + 1 - A + \frac{A}{D}x_j\right) + \left(A + \frac{(D/n)^n}{\prod_i x_i} - \frac{A}{D}\sum_i x_i - 1\right)}{\frac{1}{x_j}\left(\frac{A}{D}\sum_{i=1}^n x_i + 1 - A\right) + \frac{A}{D}} \\ &\overset{2.}{=} \frac{x_j\left(\frac{A}{D}x_j + \frac{(D/n)^n}{\prod_i x_i}\right)}{\frac{A}{D}\sum_i x_i + 1 - A + \frac{A}{D}x_j} \\ &\overset{3.}{=} \frac{x_j^2 + \frac{D}{A}\frac{(D/n)^n}{\prod_{i\neq j} x_i}}{\sum_i x_i + \frac{D}{A} - D + x_j} \end{aligned}$$

1. substitute, absorb minus sign into denominator, and convert to same denominator
2. simplify numerator and expand by x_j
3. expand by $\frac{D}{A}$

We verify with Curve's 3pool code and find that things check out:

$$\begin{aligned} y &= x_j \\ c &= \frac{D}{An^n}\frac{(D/n)^n}{\prod_{i\neq j} x_i} \\ b &= \sum_{i\neq j} x_i + \frac{D}{An^n} \\ y_{\text{next}} &= \frac{y^2 + c}{2y + b - D} \\ &= \frac{x_j^2 + \frac{D}{An^n}\frac{(D/n)^n}{\prod_{i\neq j} x_i}}{2x_j + \sum_{i\neq j} x_i + \frac{D}{An^n} - D} \end{aligned}$$

Even though $\frac{D}{n}$ seems like the canonical choice for the initial guess of x_j, we actually have to use $\frac{D}{2}$. To see why, look at $G(x_j)$ and consider a pool that's arbitrarily far from equilibrium (i.e. $\forall i \neq j : x_i < \epsilon$). The numerator of $G(x_j)$ would grow arbitrarily large, while the denominator would tend towards $2x_j + \frac{D}{A} - D$ and would hence be 0 or even negative if $2x_j \leq D - \frac{D}{A}$. Since A is an arbitrary parameter, $\frac{D}{A}$ can be arbitrarily small, therefore we have to choose $x_j = \frac{D}{2}$ to ensure a sensible value of the denominator at all times.

Finding D

We perform an essentially equivalent series of steps as we did for x_j to find D.

$$\begin{aligned} F(D) &= AD + D\frac{(D/n)^n}{\prod_{i=1}^n x_i} - A\sum_{i=1}^n x_i - D \\ F'(D) &= A + (n+1)\frac{(D/n)^n}{\prod_{i=1}^n x_i} - 1 \end{aligned}$$

Again, using Newton's method:

$$\begin{aligned} G(D) &= D - \frac{F(D)}{F'(D)} \\ &= \frac{\left(AD + (n+1)D\frac{(D/n)^n}{\prod_i x_i} - D\right) - \left(AD + D\frac{(D/n)^n}{\prod_i x_i} - A\sum_i x_i - D\right)}{A + (n+1)\frac{(D/n)^n}{\prod_i x_i} - 1} \\ &= \frac{nD\frac{(D/n)^n}{\prod_i x_i} + A\sum_i x_i}{A + (n+1)\frac{(D/n)^n}{\prod_i x_i} - 1} \end{aligned}$$

Using $D_0 = \sum_i x_i$ for a pool in equilibrium, we can then iterate $D_{m+1} = G(D_m)$ until it converges to G's fixed point D (which again checks out with Curve's 3pool code).

Plotting x_j as a function of x_i (where all other $x_k, k \neq i, k \neq j$ are set to their equilibrium value $\frac{D}{n}$):

In [17]:

```python
interact_2d('stableswap')
```

n 2

D 2.00

☐ zoom

☐ equal_aspect

fixed_axis 0

resolution 50

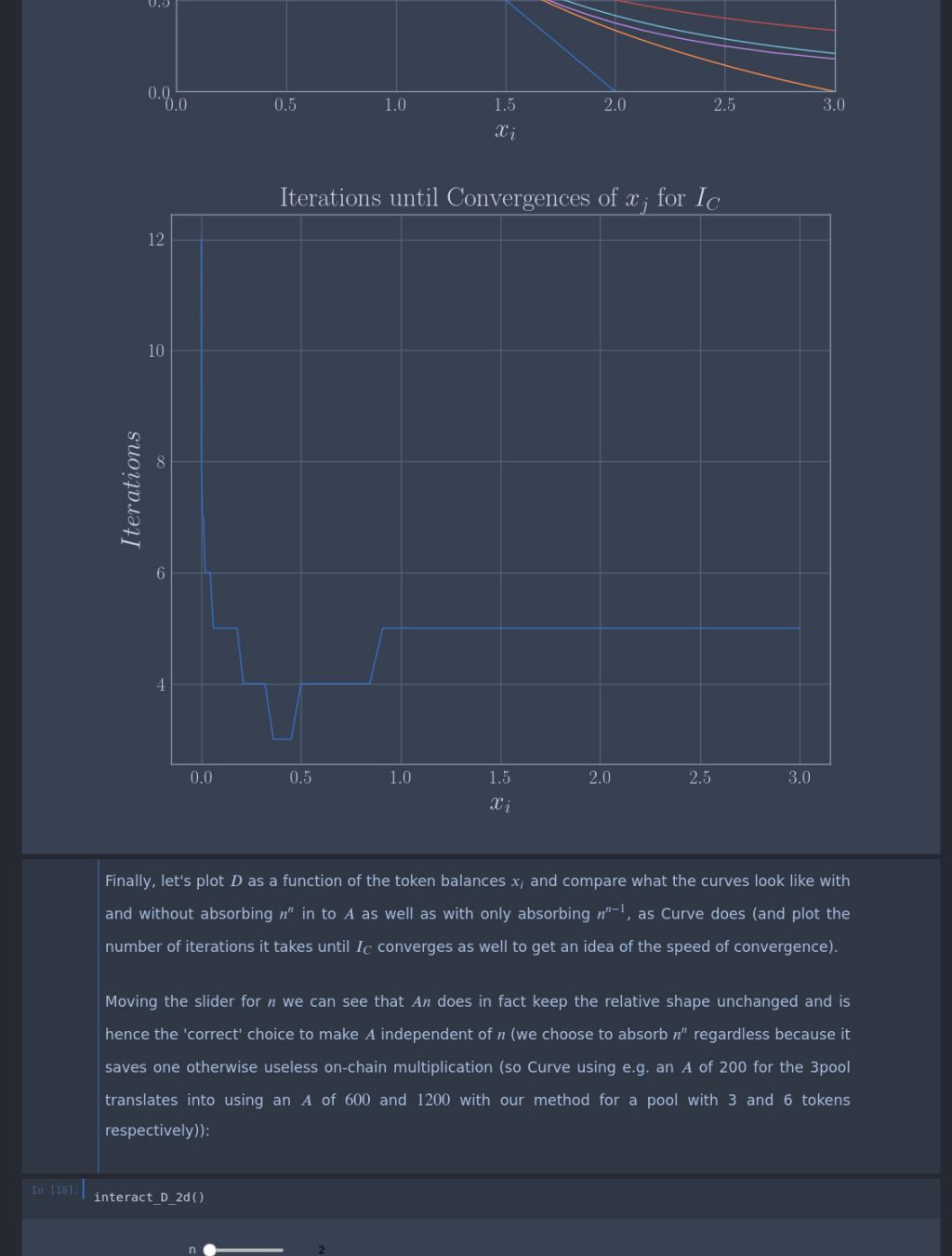
chi 1.00

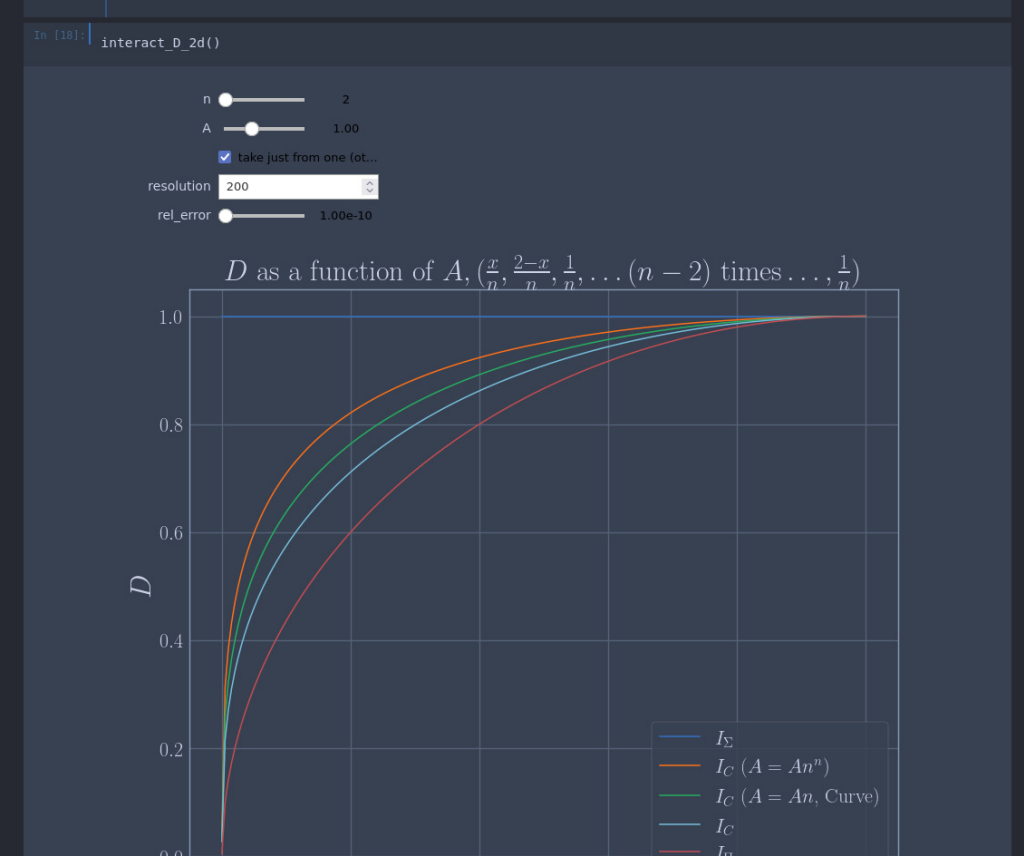
☑ with_theta

theta 0.50

A 1.00

rel_error 1.00e-10





Finally, let's plot D as a function of the token balances x_i and compare what the curves look like with and without absorbing n^n in to A as well as with only absorbing n^{n-1}, as Curve does (and plot the number of iterations it takes until I_C converges as well to get an idea of the speed of convergence).

Moving the slider for n we can see that An does in fact keep the relative shape unchanged and is hence the 'correct' choice to make A independent of n (we choose to absorb n^n regardless because it saves one otherwise useless on-chain multiplication (so Curve using e.g. an A of 200 for the 3pool translates into using an A of 600 and 1200 with our method for a pool with 3 and 6 tokens respectively)):

In [18]:

```python
interact_D_2d()
```

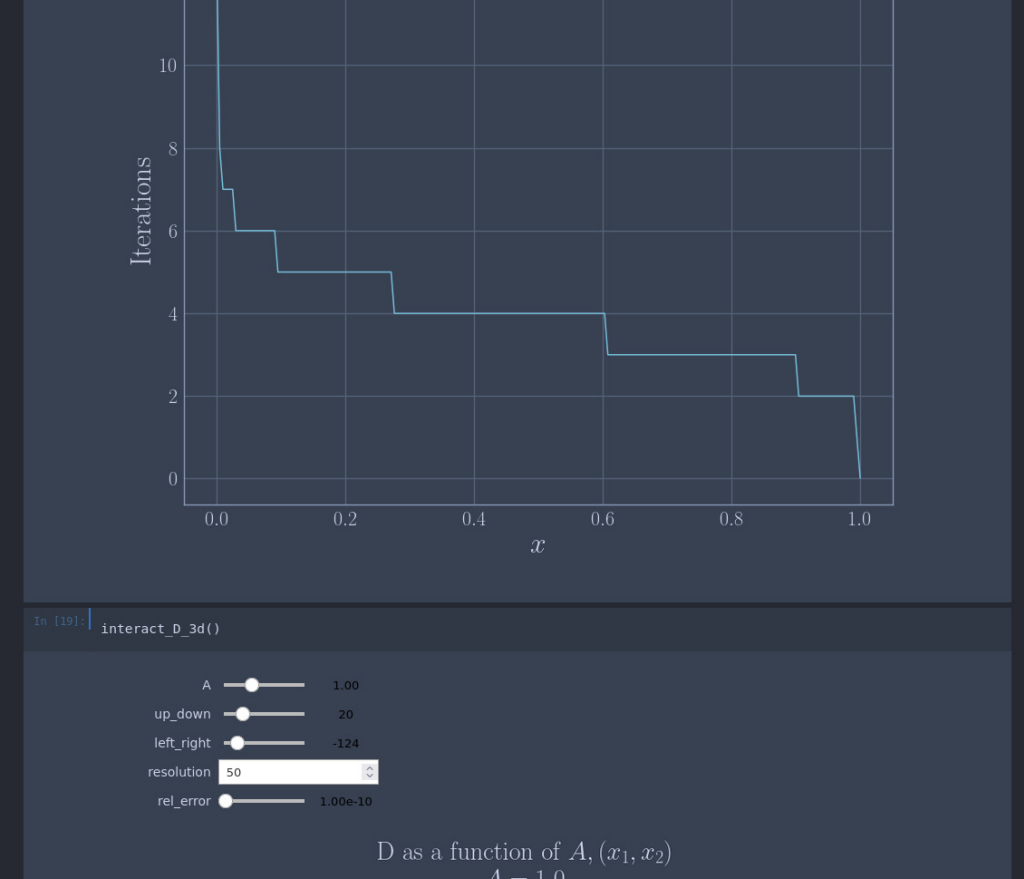
n 2

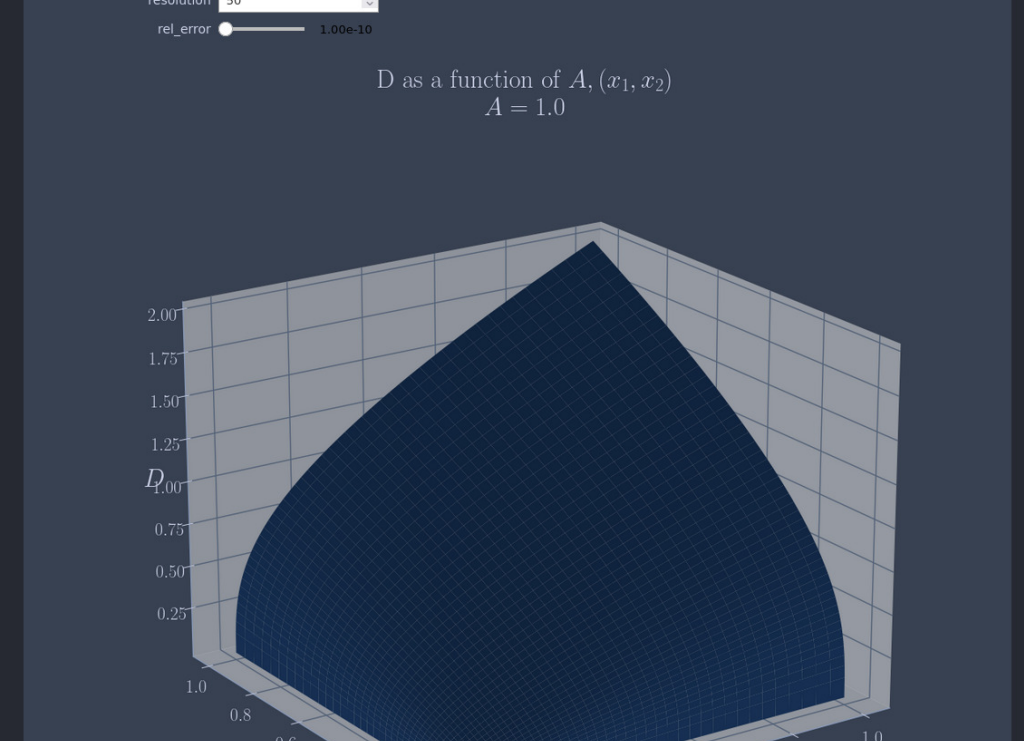
A 1.00

☑ take just from one (ot...

resolution 200

rel_error 1.00e-10





In [19]:

```python
interact_D_3d()
```

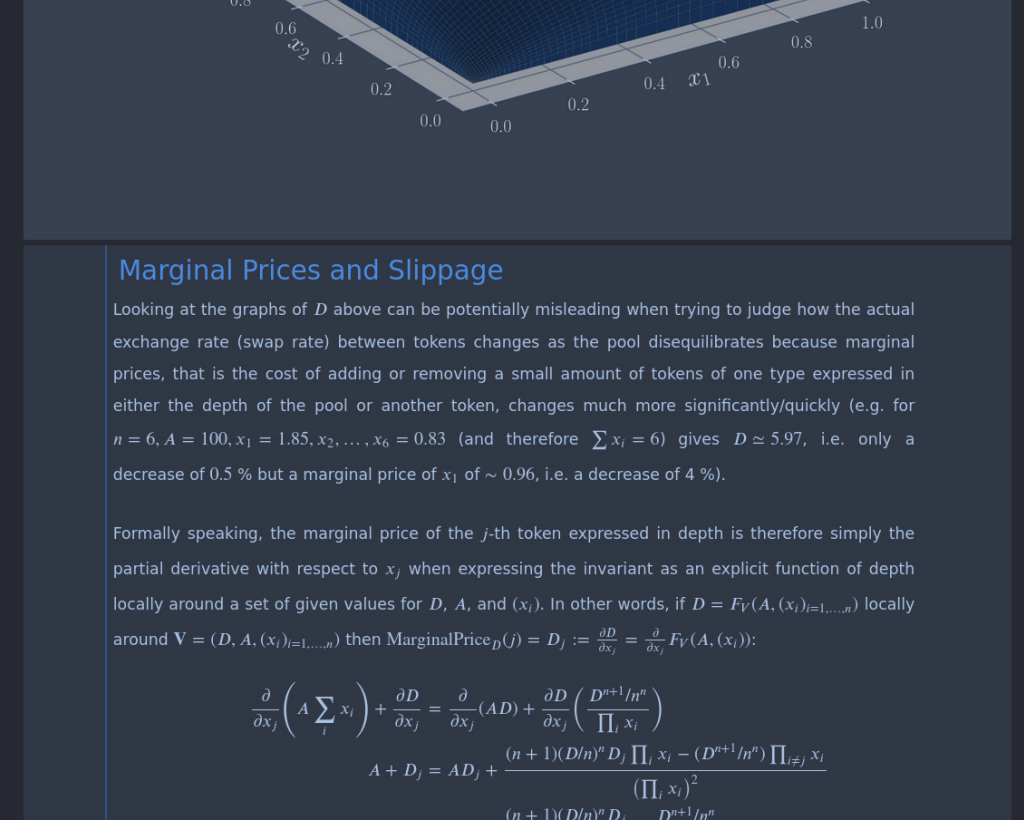
A 1.00

up_down 20

left_right -124

resolution 50

rel_error 1.00e-10



Marginal Prices and Slippage

Looking at the graphs of D above can be potentially misleading when trying to judge how the actual exchange rate (swap rate) between tokens changes as the pool disequilibrates because marginal prices, that is the cost of adding or removing a small amount of tokens of one type expressed in either the depth of the pool or another token, changes much more significantly/quickly (e.g. for $n = 6, A = 100, x_1 = 1.85, x_2, \ldots, x_6 = 0.83$ (and therefore $\sum x_i = 6$) gives $D \simeq 5.97$, i.e. only a decrease of 0.5 % but a marginal price of x_1 of ~ 0.96, i.e. a decrease of 4 %).

Formally speaking, the marginal price of the j-th token expressed in depth is therefore simply the partial derivative with respect to x_j when expressing the invariant as an explicit function of depth locally around a set of given values for D, A, and (x_i). In other words, if $D = F_V(A, (x_i)_{i=1,\ldots,n})$ locally around $\mathbf{V} = (D, A, (x_i)_{i=1,\ldots,n})$ then $\text{MarginalPrice}_D(j) = D_j := \frac{\partial D}{\partial x_j} = \frac{\partial}{\partial x_j}F_V(A, (x_i))$:

$$\frac{\partial}{\partial x_j}\left(A\sum_i x_i\right) + \frac{\partial D}{\partial x_j} = \frac{\partial}{\partial x_j}(AD) + \frac{\partial D}{\partial x_j}\left(\frac{D^{n+1}/n^n}{\prod_i x_i}\right)$$

$$A + D_j = AD_j + \frac{(n+1)(D/n)^n D_j \prod_i x_i - (D^{n+1}/n^n)\prod_{i\neq j} x_i}{\left(\prod_i x_i\right)^2}$$