Pairs Trading - Distance Approach

This notebook recreates a distance-based small-cap pairs trading study over 1998-2009. We build a liquid small-cap universe, normalize prices, rank spreads via squared Euclidean distance, and simulate simple mean-reversion trades. Results assume fills at the daily close, ignore commissions, and only approximate slippage, so any performance takeaway is strictly illustrative.

Table of contents

Universe Construction Overview

This next cell spins up a `QuantBook`, sets the historical window (1998-2009), and defines coarse/fine filters to capture the most liquid U.S. small caps. We preselect liquid names via dollar volume, then clamp market caps to a small-cap band so the downstream pair analysis focuses on similarly scaled issuers.

In [2]:
```
from collections import defaultdict
from datetime import datetime
import pandas as pd
from QuantConnect import Resolution
from QuantConnect.Research import QuantBook

qb = QuantBook()

START_DATE = datetime(1998, 1, 1)
END_DATE = datetime(2009, 12, 31)
SMALL_CAP_MIN = 3e8
SMALL_CAP_MAX = 2e9
TARGET_COUNT = 1000
```

```
qb.SetStartDate(END_DATE)
qb.SetEndDate(END_DATE)
qb.UniverseSettings.Resolution = Resolution.Daily

coarse_dollar_volume = defaultdict(float)

def coarse_selection(coarse):
    coarse_dollar_volume.clear()
    filtered = [
        c for c in coarse
        if c.HasFundamentalData
        and c.Price and c.Price > 0
        and c.DollarVolume and c.DollarVolume > 0
    ]
    for c in filtered:
        coarse_dollar_volume[c.Symbol] = c.DollarVolume
    return [c.Symbol for c in sorted(filtered, key=lambda c: c.DollarVolume, revers

def fine_selection(fine):
    eligible_small_caps = [
        f for f in fine
        if f.MarketCap
        and SMALL_CAP_MIN <= f.MarketCap <= SMALL_CAP_MAX
    ]
    ranked = sorted(
        eligible_small_caps,
        key=lambda f: coarse_dollar_volume.get(f.Symbol, 0),
        reverse=True
    )
    return [f.Symbol for f in ranked[:TARGET_COUNT]]

universe = qb.AddUniverse(coarse_selection, fine_selection)

universe_history = qb.UniverseHistory(universe, START_DATE, END_DATE).droplevel([0]
print("Snapshots captured:", len(universe_history))
print(universe_history.apply(len).describe())
```

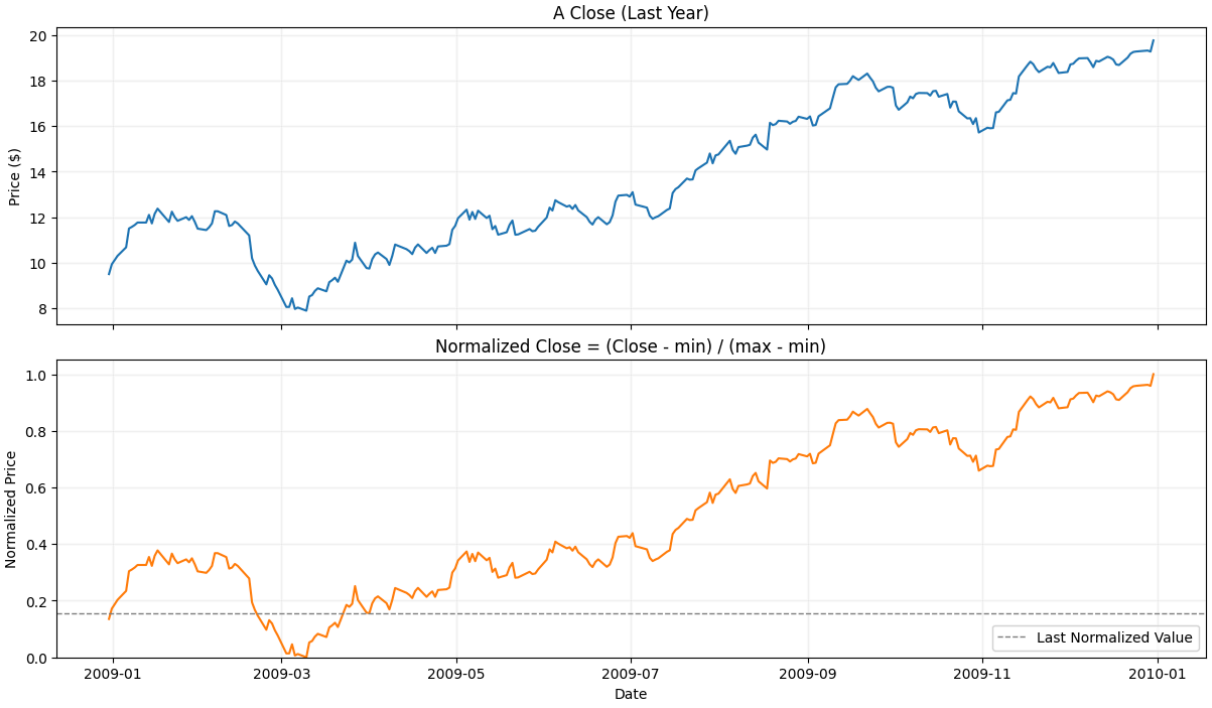
```
Snapshots captured: 3018
count    3018.000000
mean     2755.987409
std       280.874110
min      2093.000000
25%      2562.000000
50%      2908.500000
75%      3000.000000
max      3000.000000
Name: data, dtype: float64
```

Price Normalization Setup

Here we fetch one year of daily history for every symbol captured by the universe snapshots, stitch the batches together, and normalize closes via $(\text{last close} - \min)/(\max - \min)$. The illustrative plot double-checks that the transformation behaves as expected on a sample constituent.

In [3]:
```
from datetime import timedelta
import matplotlib.pyplot as plt

LOOKBACK_DAYS = 365
history_start = START_DATE - timedelta(days=LOOKBACK_DAYS)
BATCH_SIZE = 125

universe_symbols = sorted(
    {member.Symbol for datum in universe_history.values.flatten() for member in dat
 )

if not universe_symbols:
    raise ValueError("Universe history returned no symbols to normalize.")

def chunked(sequence, size):
    for idx in range(0, len(sequence), size):
        yield sequence[idx : idx + size]

close_frames = []
for symbols_chunk in chunked(universe_symbols, BATCH_SIZE):
    history_chunk = qb.History(
        symbols_chunk,
        history_start,
        END_DATE,
        Resolution.Daily
)
    if history_chunk.empty:
        continue
    close_wide = history_chunk.close.unstack(0)
    close_frames.append(close_wide)

if not close_frames:
    raise ValueError("History request returned no data across all batches.")

close_history = (
    pd.concat(close_frames, axis=1)
    .sort_index()
    .ffill()
)
close_history = close_history.loc[:, ~close_history.columns.duplicated()]
close_history = close_history.dropna(axis=1, how="all")

if close_history.empty:
    raise ValueError("Close history is empty after cleaning. Adjust the universe or

rolling_min = close_history.min()
rolling_max = close_history.max()
denominator = (rolling_max - rolling_min).replace(0, pd.NA)
normalized_last = (close_history.iloc[-1] - rolling_min) / denominator
normalized_last = normalized_last.dropna()

if normalized_last.empty:
    raise ValueError("No symbols have a valid normalized value on the last date.")

plot_window_start = close_history.index.max() - timedelta(days=LOOKBACK_DAYS)
```

```
example_window = close_history.loc[plot_window_start:]

if example_window.empty:
    raise ValueError("Plot window is empty; cannot build illustration chart.")

example_symbol = normalized_last.index[0]
example_close = example_window[example_symbol].dropna()
example_min = example_close.min()
example_max = example_close.max()
if example_max == example_min:
    raise ValueError(f"Symbol {example_symbol.Value} had no price variation during
example_normalized = (example_close - example_min) / (example_max - example_min)

fig, axes = plt.subplots(2, 1, figsize=(12, 7), sharex=True, constrained_layout=Tru
axes[0].plot(example_close.index, example_close.values, color="#1f77b4")
axes[0].set_title(f"{example_symbol.Value} Close (Last Year)")
axes[0].set_ylabel("Price ($)")
axes[0].grid(True, alpha=0.15)

axes[1].plot(example_normalized.index, example_normalized.values, color="#ff7f0e")
axes[1].axhline(normalized_last.loc[example_symbol], color="gray", linestyle="--",
axes[1].set_title("Normalized Close = (Close - min) / (max - min)")
axes[1].set_ylabel("Normalized Price")
axes[1].set_xlabel("Date")
axes[1].set_ylim(0, 1.05)
axes[1].grid(True, alpha=0.15)
axes[1].legend(loc="lower right")

plt.show()

print(
    f"Normalized price for {example_symbol.Value} as of {close_history.index[-1].da
    f"{normalized_last.loc[example_symbol]:.4f}",
)
```



```
Normalized price for A as of 2009-12-30: 0.1563
```

Six-Month Pair Scan

We examine every January 1 and July 1 anchor, normalize the prior year of prices, and keep the 20 symbol pairs with the smallest squared Euclidean spread per period.

Pair Ranking Logic

The next code block slides a 365-day window for every Jan/Jul anchor, normalizes each symbol in that window, and computes squared Euclidean distances for all symbol combinations. We then cache the normalized panels plus raw prices so later steps can reuse the same data without recomputation.

The distance metric averages over every timestamp t in the window:

$$d_{ij} = \sum_{t=1}^{T} \left(z_{i,t} - z_{j,t}\right)^2 = \|z_i - z_j\|_2^2,$$

where $z_{i,t}$ is the normalized price of symbol i at time t. Smaller d_{ij} values imply tighter co-movement and therefore higher priority in the candidate list.

In [4]:
```python
from IPython.display import display
import numpy as np
from datetime import timedelta

LOOKBACK_DAYS = 365
ANCHOR_MONTH_DAY = [(1, 1), (7, 1)]
TOP_PAIR_COUNT = 20

if "close_history" not in globals():
    raise ValueError("Run the normalization cell first to populate close_history.")

prices = (
    close_history
    .sort_index()
    .ffill()
    .dropna(axis=1, how="all")
 )

if prices.empty:
    raise ValueError("close_history contains no usable data; rerun the prior cell."

trading_index = prices.index
first_available = trading_index.min()
last_available = trading_index.max()

def pad_or_skip(ts):
    """Return latest trading day on/ before ts or None if none exists."""
    pos = trading_index.searchsorted(ts, side="right") - 1
    if pos < 0:
```

```
        return None
    return trading_index[pos]

anchor_dates = []
for year in range(first_available.year, last_available.year + 1):
    for month, day in ANCHOR_MONTH_DAY:
        candidate = pd.Timestamp(year=year, month=month, day=day)
        if candidate > last_available:
            continue
        anchor_label = pad_or_skip(candidate)
        if anchor_label is None:
            continue
        window_start = anchor_label - timedelta(days=LOOKBACK_DAYS)
        if window_start < first_available:
            continue
        anchor_dates.append(anchor_label)

anchor_dates = sorted(set(anchor_dates))

if not anchor_dates:
    raise ValueError("No anchors had a full trailing window in close_history.")

top_pairs_by_period = {}
normalized_cache = {}
window_price_cache = {}

for anchor_label in anchor_dates:
    window_start = anchor_label - timedelta(days=LOOKBACK_DAYS)
    window_prices = prices.loc[window_start:anchor_label]
    if len(window_prices) < 60:
        continue

    rolling_min = window_prices.min()
    rolling_max = window_prices.max()
    denominator = (rolling_max - rolling_min).replace(0, np.nan)
    normalized_window = (window_prices - rolling_min) / denominator
    normalized_window = normalized_window.dropna(axis=1, how="any")

    if normalized_window.shape[1] < 2:
        continue

    values = normalized_window.to_numpy(dtype=float)
    col_norms = np.sum(values ** 2, axis=0)
    gram = values.T @ values
    dist_sq = col_norms[:, None] + col_norms[None, :] - 2 * gram

    n_cols = dist_sq.shape[0]
    triu_rows, triu_cols = np.triu_indices(n_cols, k=1)
    if triu_rows.size == 0:
        continue

    distances = dist_sq[triu_rows, triu_cols]
    top_k = min(TOP_PAIR_COUNT, distances.size)
    top_idx = np.argpartition(distances, top_k - 1)[:top_k]
    ordering = top_idx[np.argsort(distances[top_idx])]
```

```
    symbols = normalized_window.columns.to_list()
    pairs = []
    for idx in ordering:
        left = symbols[triu_rows[idx]]
        right = symbols[triu_cols[idx]]
        pairs.append({
            "symbol_a": left,
            "symbol_b": right,
            "distance": float(distances[idx]),
        })

    if pairs:
        top_pairs_by_period[anchor_label] = pairs
        normalized_cache[anchor_label] = normalized_window
        window_price_cache[anchor_label] = window_prices

if not top_pairs_by_period:
    raise ValueError("No qualifying pairs were identified for the available anchors

summary_rows = [{
    "anchor_end": anchor.date(),
    "window_start": (anchor - timedelta(days=LOOKBACK_DAYS)).date(),
    "pair_count": len(pairs),
} for anchor, pairs in top_pairs_by_period.items()]
summary_df = pd.DataFrame(summary_rows).sort_values("anchor_end")
display(summary_df.tail())

latest_anchor = max(top_pairs_by_period)
print(
    f"Most recent anchor {latest_anchor.date()} has "
    f"{len(top_pairs_by_period[latest_anchor])} candidate pairs.",
)
display(pd.DataFrame(top_pairs_by_period[latest_anchor]).head())
```

	anchor_end	window_start	pair_count
16	2007-06-29	2006-06-29	20
17	2007-12-31	2006-12-31	20
18	2008-06-30	2007-07-01	20
19	2008-12-31	2008-01-01	20
20	2009-06-30	2008-06-30	20

```
Most recent anchor 2009-06-30 has 20 candidate pairs.
```

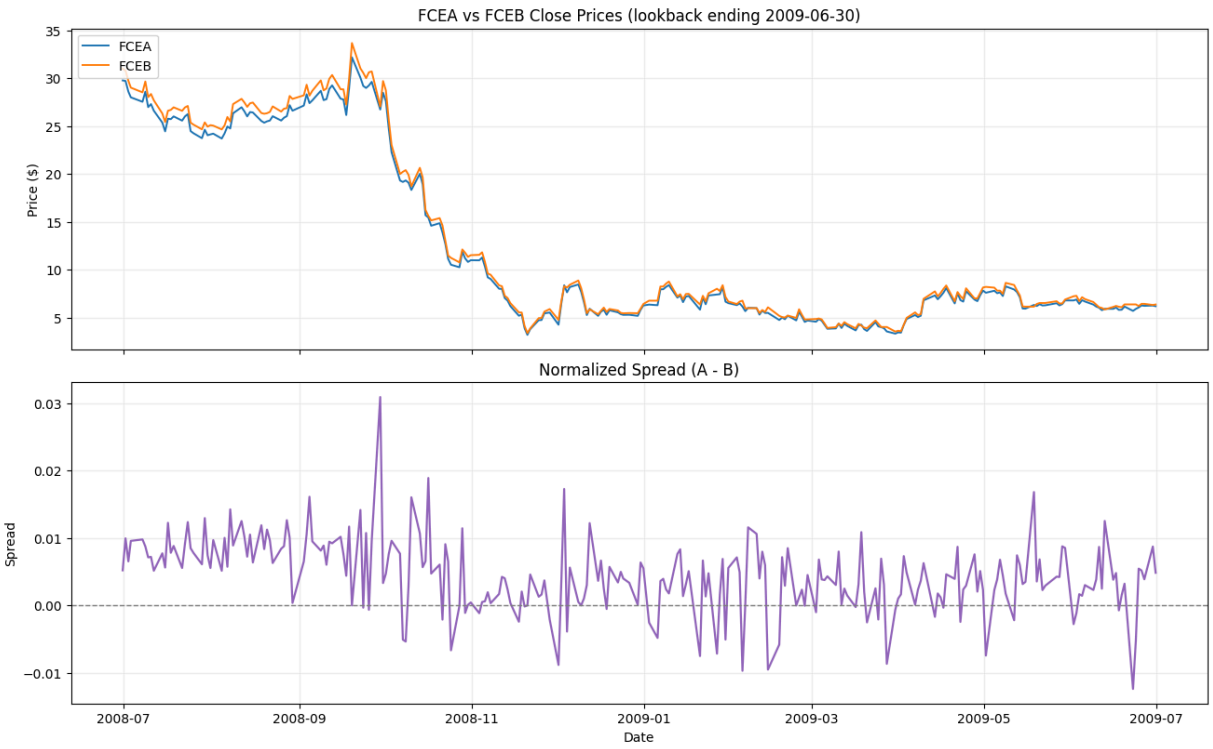
	symbol_a	symbol_b	distance
0	FCEA R735QTJ8XC9X	FCEB R735QTJ8XC9X	0.012115
1	GLO TI4IYO3P7VJ9	GLQ T8648MY9D945	0.043434
2	LBTYA SZC2UFSQNK9X	LBTYK TBT2P2LPJLGL	0.064653
3	BRKA R735QTJ8XC9X	BRKB R735QTJ8XC9X	0.111270
4	GLQ T8648MY9D945	GLV T0PB7436HAAT	0.162386

Visual Inspection Of A Candidate Pair

To sanity check the rankings, the next cell pulls the top-ranked pair from the latest anchor and plots raw prices plus the normalized spread so we can confirm the distance metric is capturing reasonably co-moving legs.

In [5]:
```
import matplotlib.pyplot as plt

if not top_pairs_by_period:
    raise ValueError("Run the pair-selection cell to populate the pair dictionary f

latest_anchor = max(top_pairs_by_period)
sample_pair = top_pairs_by_period[latest_anchor][0]
symbol_a = sample_pair["symbol_a"]
symbol_b = sample_pair["symbol_b"]

window_prices = window_price_cache[latest_anchor][[symbol_a, symbol_b]].dropna()
normalized_window = normalized_cache[latest_anchor][[symbol_a, symbol_b]].dropna()
spread_series = normalized_window[symbol_a] - normalized_window[symbol_b]

fig, axes = plt.subplots(2, 1, figsize=(13, 8), sharex=True, constrained_layout=Tru
axes[0].plot(window_prices.index, window_prices[symbol_a], label=symbol_a.Value, li
axes[0].plot(window_prices.index, window_prices[symbol_b], label=symbol_b.Value, li
axes[0].set_title(
    f"{symbol_a.Value} vs {symbol_b.Value} Close Prices (lookback ending {latest_an
 )
axes[0].set_ylabel("Price ($)")
axes[0].grid(True, alpha=0.2)
axes[0].legend(loc="upper left")

axes[1].plot(spread_series.index, spread_series, color="#9467bd", linewidth=1.6)
axes[1].axhline(0, color="gray", linestyle="--", linewidth=1)
axes[1].set_title("Normalized Spread (A - B)")
axes[1].set_ylabel("Spread")
axes[1].set_xlabel("Date")
axes[1].grid(True, alpha=0.2)

plt.show()

print(
    f"Sample pair ending {latest_anchor.date()}: "
```

```
    f"{symbol_a.Value}/{symbol_b.Value} squared distance = {sample_pair['distance']
)
```




```
Sample pair ending 2009-06-30: FCEA/FCEB squared distance = 0.012115
```

Spread Volatility Ranking

With the most recent top 20 pairs in hand, we compute the standard deviation of each normalized spread so we can prioritize tighter, likely mean-reverting relationships before feeding them into the trade simulator.

In [6]:
```
if not top_pairs_by_period:
    raise ValueError("Run the pair-selection cell first to build the pair dictionar

latest_anchor = max(top_pairs_by_period)
normalized_window = normalized_cache[latest_anchor]
pairs = top_pairs_by_period[latest_anchor]

vol_rows = []
for pair in pairs:
    symbol_a = pair["symbol_a"]
    symbol_b = pair["symbol_b"]
    spread = normalized_window[symbol_a] - normalized_window[symbol_b]
    spread_std = float(spread.std(ddof=1))
    vol_rows.append({
        "symbol_a": symbol_a,
        "symbol_b": symbol_b,
        "spread_std": spread_std,
        "distance": pair["distance"],
    })

spread_vol_df = pd.DataFrame(vol_rows).sort_values("spread_std")
```

```
print(
    f"Spread volatility computed for {len(spread_vol_df)} pairs ending {latest_anch
)
display(spread_vol_df.head(10))
```

```
Spread volatility computed for 20 pairs ending 2009-06-30.
```

	symbol_a	symbol_b	spread_std	distance
0	FCEA R735QTJ8XC9X	FCEB R735QTJ8XC9X	0.005344	0.012115
2	LBTYA SZC2UFSQNK9X	LBTYK TBT2P2LPJLGL	0.012599	0.064653
1	GLO TI4IYO3P7VJ9	GLQ T8648MY9D945	0.013113	0.043434
3	BRKA R735QTJ8XC9X	BRKB R735QTJ8XC9X	0.014058	0.111270
6	NCV SNBTLE7N2CTH	NCZ SQPX0ZKKJWPX	0.020396	0.220172
8	OTCM R735QTJ8XC9X	RVT R735QTJ8XC9X	0.023916	0.252777
4	GLQ T8648MY9D945	GLV T0PB7436HAAT	0.024069	0.162386
5	GLO TI4IYO3P7VJ9	GLV T0PB7436HAAT	0.024300	0.167909
7	NBR R735QTJ8XC9X	PTEN R735QTJ8XC9X	0.026060	0.248324
11	AGU R735QTJ8XC9X	POT R735QTJ8XC9X	0.031271	0.347729

Mean-Reversion Backtest & Account Simulation

Signal & Trade Generation Logic

The upcoming code walks every anchor's top pairs, builds simple z-score style bands off the normalized spread, and simulates single-entry single-exit trades sized to $50k per leg. Trades flip direction depending on whether the spread is rich (short A / long B) or cheap (long A / short B).

In [7]:
```
ENTRY_SIGMA = 2.0
EXIT_SIGMA = 1.0
PER_TRADE_LEG_NOTIONAL = 50_000  # $50k allocated per leg (100k total per pair)

def syminfo(symbol):
    return symbol.Value if hasattr(symbol, "Value") else str(symbol)

if not top_pairs_by_period:
    raise ValueError("Run the pair-selection cell first to populate the pair dictio

trades = []
for anchor_label, pairs in sorted(top_pairs_by_period.items()):
    normalized_window = normalized_cache[anchor_label]
    window_prices = window_price_cache[anchor_label]

    for rank, pair in enumerate(pairs, start=1):
        symbol_a = pair["symbol_a"]
```

```
        symbol_b = pair["symbol_b"]

        if symbol_a not in normalized_window.columns or symbol_b not in normalized_
            continue
        if symbol_a not in window_prices.columns or symbol_b not in window_prices.c
            continue

        pair_norm = normalized_window[[symbol_a, symbol_b]].dropna()
        pair_prices = window_prices[[symbol_a, symbol_b]].dropna()
        common_index = pair_norm.index.intersection(pair_prices.index)
        if len(common_index) < 10:
            continue

        spread_series = pair_norm.loc[common_index, symbol_a] - pair_norm.loc[commo
        spread_std = float(spread_series.std(ddof=1))
        if not np.isfinite(spread_std) or spread_std == 0:
            continue

        entry_upper = ENTRY_SIGMA * spread_std
        entry_lower = -ENTRY_SIGMA * spread_std
        exit_upper = EXIT_SIGMA * spread_std
        exit_lower = -EXIT_SIGMA * spread_std

        position = None
        for idx, dt in enumerate(common_index):
            spread = spread_series.loc[dt]
            price_a = pair_prices.loc[dt, symbol_a]
            price_b = pair_prices.loc[dt, symbol_b]

            if position is None:
                if spread >= entry_upper:
                    shares_a = PER_TRADE_LEG_NOTIONAL / price_a
                    shares_b = PER_TRADE_LEG_NOTIONAL / price_b
                    position = {
                        "direction": "short_spread",
                        "entry_dt": dt,
                        "entry_spread": spread,
                        "entry_price_a": price_a,
                        "entry_price_b": price_b,
                        "shares_a": shares_a,
                        "shares_b": shares_b,
                    }
                elif spread <= entry_lower:
                    shares_a = PER_TRADE_LEG_NOTIONAL / price_a
                    shares_b = PER_TRADE_LEG_NOTIONAL / price_b
                    position = {
                        "direction": "long_spread",
                        "entry_dt": dt,
                        "entry_spread": spread,
                        "entry_price_a": price_a,
                        "entry_price_b": price_b,
                        "shares_a": shares_a,
                        "shares_b": shares_b,
                    }
                continue
```

```
            # Manage exit conditions
            should_close = exit_lower <= spread <= exit_upper
            is_last_bar = idx == len(common_index) - 1
            if should_close or is_last_bar:
                if position["direction"] == "short_spread":
                    pnl = (-position["shares_a"] * (price_a - position["entry_price
                    pnl += position["shares_b"] * (price_b - position["entry_price_
                else:
                    pnl = position["shares_a"] * (price_a - position["entry_price_a
                    pnl += (-position["shares_b"] * (price_b - position["entry_pric

                trades.append({
                    "anchor_end": anchor_label.date(),
                    "pair_rank": rank,
                    "symbol_a": syminfo(symbol_a),
                    "symbol_b": syminfo(symbol_b),
                    "direction": position["direction"],
                    "entry_date": position["entry_dt"],
                    "exit_date": dt,
                    "entry_spread": position["entry_spread"],
                    "exit_spread": spread,
                    "spread_std": spread_std,
                    "holding_days": (dt - position["entry_dt"]).days or 1,
                    "pnl": pnl,
                })
                position = None

trade_log = pd.DataFrame(trades)
if trade_log.empty:
    raise ValueError("No trades were generated; consider loosening thresholds or ve

trade_log = trade_log.sort_values(["exit_date", "pair_rank"]).reset_index(drop=True
print(f"Generated {len(trade_log)} trades across {trade_log.anchor_end.nunique()} a
display(trade_log.head(10))
```

```
Generated 2287 trades across 21 anchor periods.
```

	anchor_end	pair_rank	symbol_a	symbol_b	direction	entry_date	exit_date	entry_spr
0	1999-06-30	20	TESOF	Z	long_spread	1998-07-06 16:00:00	1998-07-07 16:00:00	-0.190
1	1999-06-30	3	CNU	SVT	long_spread	1998-07-06 16:00:00	1998-07-08 16:00:00	-0.122
2	1999-06-30	11	ASTX	TXN	short_spread	1998-06-30 16:00:00	1998-07-10 16:00:00	0.134
3	1999-06-30	1	BFA	BFB	long_spread	1998-07-06 16:00:00	1998-07-29 16:00:00	-0.078
4	1999-06-30	8	WSO	WSOB	long_spread	1998-07-27 16:00:00	1998-07-30 16:00:00	-0.134
5	1999-06-30	20	TESOF	Z	short_spread	1998-07-24 16:00:00	1998-07-30 16:00:00	0.139
6	1999-06-30	8	WSO	WSOB	long_spread	1998-08-03 16:00:00	1998-08-04 16:00:00	-0.131
7	1999-06-30	11	ASTX	TXN	short_spread	1998-08-11 16:00:00	1998-08-18 16:00:00	0.131
8	1999-06-30	9	AMAT	TER	short_spread	1998-08-05 16:00:00	1998-08-24 16:00:00	0.131
9	1999-06-30	8	WSO	WSOB	long_spread	1998-08-25 16:00:00	1998-08-26 16:00:00	-0.153

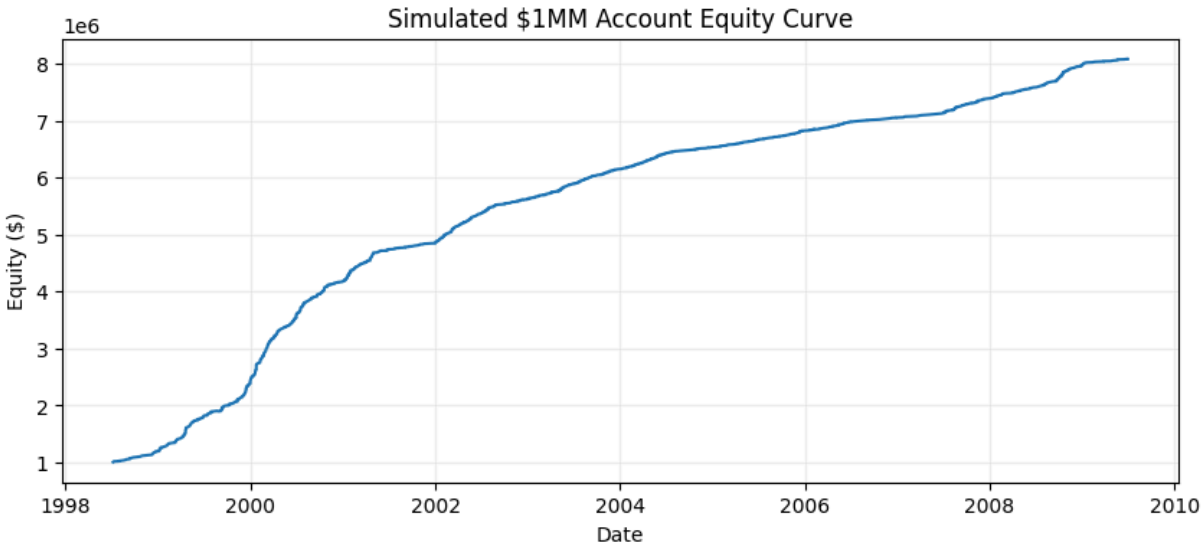
Equity Simulation Caveats

We aggregate trade-level PnL into a notional $1MM book to visualize the equity curve. This analysis still ignores explicit broker commissions, models only a placeholder slippage figure, and assumes fills at the daily close, so the performance is optimistic relative to what live execution would deliver.

In [9]:
```
ACCOUNT_START = 1_000_000
SLIPPAGE_BPS = 1  # assume 1 bp per leg per trade

if "trade_log" not in globals() or trade_log.empty:
```

```
    raise ValueError("Run the backtest cell to generate the trade log first.")

trade_log = trade_log.copy()
trade_log["slippage_cost"] = 2 * SLIPPAGE_BPS / 10_000 * PER_TRADE_LEG_NOTIONAL
trade_log["net_pnl"] = trade_log["pnl"] - trade_log["slippage_cost"]

equity = ACCOUNT_START
equity_curve = []
for _, row in trade_log.sort_values("exit_date").iterrows():
    equity += row["net_pnl"]
    equity_curve.append({"date": row["exit_date"], "equity": equity})

equity_curve_df = pd.DataFrame(equity_curve)
ending_equity = equity_curve_df["equity"].iloc[-1] if not equity_curve_df.empty els
summary = {
    "Starting Equity": ACCOUNT_START,
    "Ending Equity": ending_equity,
    "Net Return %": (ending_equity / ACCOUNT_START - 1) * 100,
    "Trades": len(trade_log),
    "Win Rate %": (trade_log["net_pnl"] > 0).mean() * 100,
    "Avg Net PnL": trade_log["net_pnl"].mean(),
    "Max Drawdown": (equity_curve_df["equity"].cummax() - equity_curve_df["equity"]
}

summary_df = pd.DataFrame([summary])
display(summary_df)

plt.figure(figsize=(10, 4))
plt.plot(equity_curve_df["date"], equity_curve_df["equity"], color="#1f77b4")
plt.title("Simulated $1MM Account Equity Curve")
plt.xlabel("Date")
plt.ylabel("Equity ($)")
plt.grid(True, alpha=0.2)
plt.show()
```

	Starting Equity	Ending Equity	Net Return %	Trades	Win Rate %	Avg Net PnL	Max Drawdown
0	1000000	8.080937e+06	708.093689	2287	98.38216	3096.168293	8665.206299



Method Review & Next Experiments

Upsides. The workflow is transparent, easy to recompute in QuantBook, and produces interpretable normalized spreads so we can reason about each candidate pair before trading. It also provides a fast baseline for gauging how sensitive a distance-based selection is to anchor timing. **Downsides.** Performance is overstated because we assume close-only fills, ignore explicit commissions, and rely on a crude per-trade slippage proxy. The distance metric can overweight transient co-movements, and the single-entry/single-exit structure leaves capital idle whenever spreads drift but fail to touch the hard sigma bands. **Variation ideas.**

- Re-tune the training window (e.g., 90/180/540 days) or the rebalance cadence before refreshing pairs.
- Adjust the entry/exit sigma thresholds or allow dynamic bands tied to realized spread variance.
- Apply better pair or asset weighting, such as volatility scaling or optimization across all open spreads.
- Swap the distance metric for spread z-scores, half-life estimates, co-integration tests, or Pearson correlations.
- Favor spreads that cross 0 frequently, or expand from strict pairs into quasi-multivariate baskets built around sectors or factor clusters. These extensions pave the way toward a cointegration-based approach, which will be the focus of the next notebook installment.