/calculation/calculator.py

```python
import numpy
import inspect
from models import binomial_tree
from models import monte_carlo
from models import pde
from securities import options
from securities import bonds
from securities import converts
from securities import analytics
from utils import pricing_utils


def get_vol_term_structure(model, tenors):
    """
    ATMF vol term structure.
    """
    vols = {}

    for tenor in tenors:
        if tenor > model.max_years:
            continue

        # Look up the time index
        t_index = numpy.searchsorted(model.time_grid, tenor)

        # Calculate the forward
        forward = model.stock_forward[t_index]

        # Price ATMF call option
        atm_call = options.EuropeanCall(strike=forward, maturity=tenor)
        opt_price = numpy.interp(
            model.stock_price, model.stock_grid[0], model.price(atm_call)[0]
        )

        # Back out the vol
        vols[tenor] = pricing_utils.bs_cvol(
            price=opt_price,
            S=model.stock_price,
            K=forward,
            T=tenor,
            r=model.r[t_index],
            q=model.dividend_yield,
        )

    return vols


def get_spread_term_structure(model, tenors):
    """
    Hazard rate term structure.
    """
    spreads = {}

    for tenor in tenors:
        if tenor > model.max_years:
            continue

        # Look up the time index
        t_index = numpy.searchsorted(model.time_grid, tenor)

        # Price the vanilla risky bond
        bond = bonds.Bond(
            maturity=tenor,
            face_value=1.0,
            coupon_schedule={},
            call_schedule={},
            put_schedule={},
            recovery_rate=0.0,
            recovery_type="par",
        )
        price = numpy.interp(
            model.stock_price, model.stock_grid[0], model.price(bond)[0]
        )
        # Back out the zero recovery spread
        spreads[tenor] = -numpy.log(price / model.P[t_index]) / tenor

    return spreads


def get_vol_skew(model, tenor, strikes):
    """
    Vol skew in percentage of forward.
    """
    skew = {}
```

```python
    for strike in strikes:
        # Look up the time index
        t_index = numpy.searchsorted(model.time_grid, tenor)

        # Calculate the forward
        forward = model.stock_forward[t_index]

        # Price the call option
        dollar_strike = strike * forward
        call = options.EuropeanCall(
            strike=dollar_strike,
            maturity=tenor,
        )
        opt_price = numpy.interp(
            model.stock_price, model.stock_grid[0], model.price(call)[0]
        )

        # Back out the vol
        skew[strike] = pricing_utils.bs_cvol(
            price=opt_price,
            S=model.stock_price,
            K=dollar_strike,
            T=tenor,
            r=model.r[t_index],
            q=model.dividend_yield,
        )

    return skew


def _build_model(input_obj, model_type):
    """
    Build the model class.
    """
    if model_type == "crr":
        model_cls = binomial_tree.BinomialTree
        model_params = dict(
            max_years=input_obj["max_years"],
            stock_price=input_obj["stock_price"],
            equity_vol=input_obj["equity_vol"],
            riskless_rate=input_obj["riskless_rate"],
            dividend_yield=input_obj["dividend_yield"],
            cash_dividend=input_obj["cash_dividend"],
            spline_size=input_obj["spline_size"],
            time_size=input_obj["time_size"],
            default_intensity=input_obj["default_intensity"],
            equity_to_credit=input_obj["equity_to_credit"],
        )

    elif model_type == "pde":
        model_cls = pde.PDE
        model_params = dict(
            max_years=input_obj["max_years"],
            stock_price=input_obj["stock_price"],
            equity_vol=input_obj["equity_vol"],
            riskless_rate=input_obj["riskless_rate"],
            log_discount_func=input_obj["log_discount_func"],
            dividend_yield=input_obj["dividend_yield"],
            cash_dividend=input_obj["cash_dividend"],
            time_size=input_obj["time_size"],
            space_size=input_obj["space_size"],
            min_space=input_obj["min_space"],
            max_space=input_obj["max_space"],
            default_intensity=input_obj["default_intensity"],
            equity_to_credit=input_obj["equity_to_credit"],
            calibration_target=input_obj["calibration_target"],
            calibration_scheme=input_obj["calibration_scheme"],
            pricing_scheme=input_obj["pricing_scheme"],
            pde_solver=input_obj["pde_solver"],
        )

    elif model_type == "lsmc":
        model_cls = monte_carlo.MonteCarlo
        model_params = dict(
            num_paths=input_obj["num_paths"],
            max_years=input_obj["max_years"],
            stock_price=input_obj["stock_price"],
            equity_vol=input_obj["equity_vol"],
            riskless_rate=input_obj["riskless_rate"],
            dividend_yield=input_obj["dividend_yield"],
            cash_dividend=input_obj["cash_dividend"],
            time_size=input_obj["time_size"],
            default_intensity=input_obj["default_intensity"],
            equity_to_credit=input_obj["equity_to_credit"],
            random_seed=input_obj["random_seed"],
            antithetic_variable=input_obj["antithetic_variable"],
            regression_method=input_obj["regression_method"],
```

```python
        )

    else:
        raise ValueError(f"invalid model type: {model_type}")

    # Make sure all parameters are explicitly defined
    signature = inspect.signature(model_cls.__init__)
    parameters = [
        p.name for p in signature.parameters.values()
        if p.name != "self"
    ]

    assert set(parameters) == set(model_params.keys())

    return model_cls(**model_params)


def _build_securities(input_obj):
    """
    TODO: add more.
    """
    conv_bond = converts.ConvertibleBond(
        maturity=input_obj["maturity"],
        face_value=input_obj["face_value"],
        coupon_schedule=input_obj["coupon_schedule"],
        recovery_rate=input_obj["recovery_rate"],
        recovery_type=input_obj["recovery_type"],
        conversion_schedule=input_obj["conversion_schedule"],
        call_schedule=input_obj["call_schedule"],
        call_notice_period=input_obj["call_notice_period"],
        call_cushion=input_obj["call_cushion"],
        call_cushion_prob_1m=input_obj["call_cushion_prob_1m"],
        put_schedule=input_obj["put_schedule"],
        dividend_protection=input_obj["dividend_protection"],
    )
    return conv_bond


def valuate(input_obj, model_type):
    """
    Main entry point.

    TODO: customize analytics for different securities.
    """
    model = _build_model(input_obj, model_type)
    conv_bond = _build_securities(input_obj)

    output_obj = model.price(
        conv_bond,
        analytics_to_price=[
            analytics.Price,
            analytics.DefaultProbability,
            analytics.OptionalConvProbability,
            analytics.MaturityConvProbability,
            analytics.CallRedemptionProbability,
            analytics.ForcedConvProbability,
            analytics.PutProbability,
            analytics.MaturityRedemptionProbability,
        ]
    )

    return model, conv_bond, output_obj
```

/models/binomial_tree.py

```python
import numpy
from models import model_base


class BinomialTree(model_base.ModelBase):
    def __init__(
        self,
        max_years,
        stock_price,
        equity_vol,
        riskless_rate,
        log_discount_func="cubic",
        dividend_yield=0.0,
        cash_dividend={},
        spline_size=50,
        time_size=300,
        default_intensity=0.0,
        equity_to_credit=0.0,
    ):
        """
```

```python
        Cox, Ross, Rubinstein tree.
        """
        super().__init__(
            max_years=max_years,
            stock_price=stock_price,
            equity_vol=equity_vol,
            riskless_rate=riskless_rate,
            log_discount_func=log_discount_func,
            dividend_yield=dividend_yield,
            cash_dividend=cash_dividend,
            time_size=time_size,
            default_intensity=default_intensity,
            equity_to_credit=equity_to_credit,
        )
        self.spline_size = spline_size

        self.u = numpy.exp(self.equity_vol * numpy.sqrt(self.dt))
        self.d = numpy.exp(-self.equity_vol * numpy.sqrt(self.dt))
        self.time_grid = self.build_time_grid()
        self.P = self.build_discount_curve()
        self.r = self.build_forward_rate_curve()
        # Cannot have vol TS because it would lose the recombining property.
        self.sigma = numpy.full(self.time_size, self.equity_vol)
        self.lam = numpy.full(self.time_size, self.default_intensity)
        self.cq, self.q = self.build_dividend_curve()
        self.stock_grid = self.build_stock_grid()
        self.jtd_grid = self.build_jtd_grid()
        self.pu_grid = self.build_pu_grid()


    def build_stock_grid(self):
        """
        Each entry is the stock spline of a step.

        Need to extend the tree backwards for spline_size steps.
        """
        # Initialize with current stock price and build grid iteratively
        stock_grid = [numpy.array([self.stock_price])]

        for _ in range(self.spline_size + self.time_size - 2):
            next_level = numpy.append(stock_grid[-1] * self.d, stock_grid[-1][-1] * self.u)
            stock_grid.append(next_level)

        return stock_grid[self.spline_size - 1:]


    def build_jtd_grid(self):
        """
        Each entry is the state-contigent default intensity of a step.
        """
        jtd_grid = []

        for i in range(self.time_size - 1):
            intensity = self.jtd_func(
                self.lam[i],
                self.stock_price,
                self.stock_grid[i],
                self.equity_to_credit,
            )
            jtd_grid.append(numpy.clip(intensity, None, 10))

        return jtd_grid


    def build_pu_grid(self):
        """
        Each entry is the state-contigent up-move probability of a step.

        See equation (6.77) in Handbook of Convertible Bonds.
        """
        pu_grid = []

        for i in range(self.time_size - 1):
            pu = (numpy.exp((self.r[i] + self.jtd_grid[i] - self.dividend_yield)
                * self.dt) - self.d) / (self.u - self.d)
            pu_grid.append(pu)

        return pu_grid


    def _price(self, price_obj, event_obj):
        """
        Backward diffusion.
        """
        # Start from the terminal payoff
        t_max = numpy.searchsorted(self.time_grid, price_obj.maturity)
        current_level = price_obj.payoff(self, event_obj, t_max)
```

```
        # Initialize the arrays
        continuation_array = [current_level]
        boundary_array = [current_level]
        solution_array = [current_level]

        # Step backwards
        # for t_index in range(self.time_size - 2, -1, -1):
        for t_index in range(t_max - 1, -1, -1):

            # Calculate the default probability
            discount_factor = numpy.exp(-price_obj.discount(self, t_index) * self.dt)
            prob_default = 1 - numpy.exp(-self.jtd_grid[t_index] * self.dt)

            continuation_value = (
                # Expected payoff given no default
                discount_factor
                * (1 - prob_default)
                * (current_level[1:] * self.pu_grid[t_index] + current_level[:-1] * (1 - self.pu_grid[t_index]))

                # Expected payoff given default
                + discount_factor
                * prob_default
                * price_obj.recovery(self, event_obj, t_index)

                # Coupon
                + price_obj.coupon(event_obj, t_index)
            )

            # Can directly use continuation value to compare against exercise value
            # because it is the true risk-neutral expectation of future payoffs conditioned
            # on no early exercise at current timestamp.
            exercise_boundary = continuation_value

            # The events that need to be handled depends on the security
            current_level = price_obj.handle_events(
                self,
                event_obj,
                exercise_boundary,
                continuation_value,
                t_index,
            )

            # Use piecewise lognormal method to handle discrete dividends
            # See Vellekoop & Nieuwenhuis 2005
            if self.cq[t_index] > 0:
                current_level = numpy.interp(
                    self.stock_grid[t_index] - self.cq[t_index],
                    self.stock_grid[t_index],
                    current_level,
                )

            # Keep all steps for debugging purposes
            continuation_array = [continuation_value] + continuation_array
            boundary_array = [exercise_boundary] + boundary_array
            solution_array = [current_level] + solution_array

        return continuation_array, boundary_array, solution_array
```

/models/model_base.py

```
import numpy
from securities import analytics
from objects import events
from utils import parsing_utils


class ModelBase(object):
    """
    Base class for all models.
    """
    def __init__(
        self,
        max_years,
        stock_price,
        equity_vol,
        riskless_rate,
        log_discount_func="cubic",
        dividend_yield=0.0,
        cash_dividend={},
        time_size=300,
        default_intensity=0.0,
        equity_to_credit=0.0,
    ):
        # Avoid floating point when this is set to
        # be equal to maturity of security object
```

```
        max_years += 1e-10

        self.max_years = max_years
        self.time_size = time_size
        self.dt = max_years / (time_size - 1)
        self.stock_price = stock_price
        self.equity_vol = equity_vol
        self.riskless_rate = riskless_rate
        self.log_discount_func = log_discount_func
        self.dividend_yield = dividend_yield
        self.cash_dividend = cash_dividend
        self.default_intensity = default_intensity
        self.equity_to_credit = equity_to_credit


    @staticmethod
    def jtd_func(
        default_intensity,
        stock_price,
        stock_grid,
        equity_to_credit,
    ):
        """
        Functional form of the jump-to-default intensity.
        """
        return default_intensity * (stock_price / stock_grid) ** equity_to_credit


    @staticmethod
    def jtd_deriv_func(
        default_intensity,
        stock_price,
        stock_grid,
        equity_to_credit,
    ):
        """
        Functional form of the first derivative of the
        jump-to-default intensity.
        """
        return (
            -equity_to_credit
            * stock_price / stock_grid ** 2
            * default_intensity
            * (stock_price / stock_grid) ** (equity_to_credit - 1)
        )


    def build_time_grid(self):
        """
        Each entry is the timestamp of a step.
        """
        return numpy.array(
            [self.dt * i for i in range(self.time_size)]
        )


    def build_discount_curve(self):
        """
        Each entry P(t) = exp(-R(t) * t) where R(t) is the market
        observed treasury yield. In other words, P(t) is the riskless
        PV of zero coupon bond maturing at t.
        """
        if isinstance(self.riskless_rate, (int, float)):
            return numpy.exp(-self.riskless_rate * self.time_grid)

        # Force log discount to pass through origin
        log_discount = {k: -v * k for k, v in self.riskless_rate.items()}
        log_discount[0.0] = 0.0

        # Fit a log discount curve
        log_discount_curve = parsing_utils.parse_curve(
            self.time_grid, log_discount, func=self.log_discount_func,
        )

        # Recover the discount curve
        discount_curve = numpy.exp(log_discount_curve)

        return discount_curve


    def build_forward_rate_curve(self):
        """
        Each entry r(t) is the instantaneous forward rate such that
        R(t) = int_0_t r(u) du / t and thus P(t) = exp(-int_0_t r(u) du).

        The reason for fitting discount rate curve in log space is to
        ensure the forward rate curve (derivative of log(P(t)) is smooth.
```

```python
        """
        forward_curve = numpy.zeros(self.time_size)

        # Right hand side difference in the interior
        forward_curve[:-1] = -numpy.log(self.P[1:] / self.P[:-1]) / self.dt

        # Flat interpolation at the end
        forward_curve[-1] = forward_curve[-2]

        return forward_curve


    def build_dividend_curve(self):
        """
        Short-term use cash, long-term use yield.
        """
        # Parse cash dividend
        has_dividend, _, dividend = parsing_utils.parse_schedule(
            self.time_grid,
            self.cash_dividend,
        )
        cq = numpy.where(has_dividend, dividend, 0.0)

        # Contunuous dividend
        q = numpy.full_like(self.time_grid, self.dividend_yield)

        # TODO: Add a timestamp to separate short vs long

        return cq, q


    def calculate_stock_forward(self):
        """
        Stock forward price.
        """
        stock_forward = numpy.zeros(self.time_size)

        # Equal to stock price
        stock_forward[0] = self.stock_price

        # Add the carry
        carry = numpy.cumsum(self.r[:-1] - self.dividend_yield) * self.dt
        stock_forward[1:] = self.stock_price * numpy.exp(carry)

        return stock_forward


    def price(
        self,
        security,
        analytics_to_price=[analytics.Price],
        debug_mode=False,
    ):
        """
        Main entry-point for backward diffusion.
        """
        # Move price to front because many analytics depend on it
        analytics_to_price = (
            [analytics.Price]
            + [a for a in analytics_to_price if a != analytics.Price]
        )

        result_dict = {}

        # Create events objects
        event_obj = events.Event(self, security)

        # Each analytic is a separate backward diffusion
        for analytic in analytics_to_price:

            price_obj = analytic(security, result_dict)

            continuation_array, boundary_array, solution_array = self._price(
                price_obj,
                event_obj,
            )

            result_dict[analytic.to_string()] = {
                "continuation": continuation_array,
                "boundary": boundary_array,
                "solution": solution_array,
            }

        # Discard the continuaton value if not needed for debugging
        if not debug_mode:
            result_dict = {k: v["solution"] for k, v in result_dict.items()}
```

```
        # Return a single result if only one analytic is specified
        return list(result_dict.values())[0] if len(result_dict) == 1 else result_dict
```

/models/monte_carlo.py

```
import numpy
from models import model_base


class MonteCarlo(model_base.ModelBase):
    def __init__(
        self,
        num_paths,
        max_years,
        stock_price,
        equity_vol,
        riskless_rate,
        log_discount_func="cubic",
        dividend_yield=0.0,
        cash_dividend={},
        time_size=300,
        default_intensity=0.0,
        equity_to_credit=0.0,
        random_seed=42,
        antithetic_variable=False,
        regression_method="all",
    ):
        """
        Longstaff & Schwartz method.
        """
        super().__init__(
            max_years=max_years,
            stock_price=stock_price,
            equity_vol=equity_vol,
            riskless_rate=riskless_rate,
            log_discount_func=log_discount_func,
            dividend_yield=dividend_yield,
            cash_dividend=cash_dividend,
            time_size=time_size,
            default_intensity=default_intensity,
            equity_to_credit=equity_to_credit,
        )
        self.num_paths = num_paths
        self.random_seed = random_seed
        self.antithetic_variable = antithetic_variable
        self.regression_method = regression_method

        self.time_grid = self.build_time_grid()
        self.P = self.build_discount_curve()
        self.r = self.build_forward_rate_curve()
        self.sigma = numpy.full(self.time_size, self.equity_vol)
        self.lam = numpy.full(self.time_size, self.default_intensity)
        self.cq, self.q = self.build_dividend_curve()
        self.jtd_grid, self.stock_grid = self.simulate_paths()


    def simulate_paths(self):
        """
        Simulate stock paths.

        To handle cash dividends we stil adopt piecewise lognormal method.
        """
        # Generate standard normal random variables
        num_paths = self.num_paths
        rng = numpy.random.default_rng(seed=self.random_seed)
        gaussian = rng.normal(0, 1, (self.num_paths, self.time_size - 1))

        # Use antithetic variables
        if self.antithetic_variable:
            num_paths = num_paths * 2
            gaussian = numpy.vstack([gaussian, -gaussian])

        # Initialize default intensity grid
        jtd_grid = []

        # Initialize stock price grid
        stock_grid = [numpy.full(num_paths, self.stock_price)]

        # Simulate paths iteratively
        for i in range(self.time_size - 1):
            # Power law function with clip if equity-to-credit is nonzero
            intensity = self.jtd_func(
                self.lam[i],
                self.stock_price,
                stock_grid[i],
                self.equity_to_credit,
            )
```

```
        # Next step
        drift = (self.r[i] + intensity - self.dividend_yield -
            0.5 * self.sigma[i] ** 2) * self.dt

        diffusion = self.sigma[i] * numpy.sqrt(self.dt) * gaussian[:, i]
        stock_price = stock_grid[i] * numpy.exp(drift + diffusion)

        # Give out planned cash dividends only if it will not bankrupt them
        stock_price = numpy.where(
            stock_price - self.cq[i] > 0, stock_price - self.cq[i], stock_price
        )

        # Append to the array
        jtd_grid.append(intensity)
        stock_grid.append(stock_price)

    return jtd_grid, stock_grid


def _price(self, price_obj, event_obj):
    """
    Backward diffusion.
    """
    # Start from the terminal payoff
    t_max = numpy.searchsorted(self.time_grid, price_obj.maturity)
    current_level = price_obj.payoff(self, event_obj, t_max)

    # Initialize the arrays
    continuation_array = [current_level]
    boundary_array = [current_level]
    solution_array = [current_level]

    # Step backwards
    # for t_index in range(self.time_size - 2, -1, -1):
    for t_index in range(t_max - 1, -1, -1):

        # Calculate the default probability
        discount_factor = numpy.exp(-price_obj.discount(self, t_index) * self.dt)
        prob_default = 1 - numpy.exp(-self.jtd_grid[t_index] * self.dt)

        continuation_value = (
            # Expected payoff given no default
            discount_factor
            * (1 - prob_default)
            * current_level

            # Expected payoff given default
            + discount_factor
            * prob_default
            * price_obj.recovery(self, event_obj, t_index)

            # PV of coupon
            + price_obj.coupon(event_obj, t_index)
        )

        # Cannot directly use contination value to compare against exercise value
        # because it is path-specific, therefore need to estimate the exercise boundary
        # (unless at last step because stock prices are exactly the same).
        if t_index > 0:
            exercise_boundary = price_obj.estimate_exercise_boundary(
                self,
                event_obj,
                continuation_value,
                t_index,
            )
        else:
            exercise_boundary = continuation_value


        # The events that need to be handled depends on the security
        current_level = price_obj.handle_events(
            self,
            event_obj,
            exercise_boundary,
            continuation_value,
            t_index,
        )

        # Keep all steps for debugging purposes
        continuation_array = [continuation_value] + continuation_array
        boundary_array = [exercise_boundary] + boundary_array
        solution_array = [current_level] + solution_array

    return continuation_array, boundary_array, solution_array
```

```
/models/pde.py
import numpy
import pandas
import inspect
from scipy import sparse
from scipy.optimize import brentq
from models import model_base
from utils import pricing_utils
from utils import parsing_utils
from utils import finite_difference


class PDE(model_base.ModelBase):
    def __init__(
        self,
        max_years,
        stock_price,
        equity_vol,
        riskless_rate,
        log_discount_func="cubic",
        dividend_yield=0.0,
        cash_dividend={},
        time_size=2000,
        space_size=3000,
        min_space=5.0,
        max_space=5.0,
        default_intensity=0.0,
        equity_to_credit=0.0,
        calibration_target=[],
        calibration_scheme="rannacher",
        pricing_scheme="rannacher",
        pde_solver="thomas",
    ):
        """
        Finite difference.
        """
        super().__init__(
            max_years=max_years,
            stock_price=stock_price,
            equity_vol=equity_vol,
            riskless_rate=riskless_rate,
            log_discount_func=log_discount_func,
            dividend_yield=dividend_yield,
            cash_dividend=cash_dividend,
            time_size=time_size,
            default_intensity=default_intensity,
            equity_to_credit=equity_to_credit,
        )
        self.space_size = space_size
        self.min_space = min_space
        self.max_space = max_space
        self.calibration_target = calibration_target
        self.calibration_scheme = calibration_scheme
        self.pricing_scheme = pricing_scheme
        self.pde_solver = pde_solver

        self.X_min = numpy.log(self.stock_price / self.min_space)
        self.X_max = numpy.log(self.stock_price * self.max_space)
        self.dX = (self.X_max - self.X_min) / (self.space_size + 1)
        self.time_grid = self.build_time_grid()
        self.P = self.build_discount_curve()
        self.r = self.build_forward_rate_curve()
        self.sigma = parsing_utils.parse_curve(self.time_grid, self.equity_vol)
        self.lam = parsing_utils.parse_curve(self.time_grid, self.default_intensity)
        self.cq, self.q = self.build_dividend_curve()
        self.X, self.stock_grid = self.build_stock_grid()
        self.stock_forward = self.calculate_stock_forward()
        self.jtd_grid = self.build_jtd_grid()
        self.calibration_theta = self.build_scheme(calibration_scheme, pde_solver)
        self.pricing_theta = self.build_scheme(pricing_scheme, pde_solver)


    def build_stock_grid(self):
        """
        Equally spaced in log stock price.
        """
        X, stock_grid = [], []
        for _ in range(self.time_size):
            X_i = numpy.array([
                self.X_min + (i + 1) * self.dX for i in range(self.space_size)
            ])

            X.append(X_i)
            stock_grid.append(numpy.exp(X_i))

        return X, stock_grid
```

```python
def build_jtd_grid(self):
    """
    Each entry is the state-contigent default intensity of a step.
    """
    jtd_grid = []

    for i in range(self.time_size):
        intensity = self.jtd_func(
            self.lam[i],
            self.stock_price,
            self.stock_grid[i],
            self.equity_to_credit,
        )
        jtd_grid.append(numpy.clip(intensity, None, 10))

    return jtd_grid


def build_scheme(self, scheme, pde_solver):
    """
    Finite difference scheme.
    """
    assert scheme in ["implicit", "crank-nicolson", "rannacher"]
    assert pde_solver in ["spsolve", "splu", "thomas"]

    if scheme == "implicit":
        return finite_difference.ImplicitScheme(pde_solver)

    elif scheme == "crank-nicolson":
        return finite_difference.CrankNicolsonScheme(pde_solver)

    else:
        return finite_difference.RannacherScheme(pde_solver)


def _price(self, price_obj, event_obj):
    """
    Backward diffusion.
    """
    # Terminal condition
    t_max = numpy.searchsorted(self.time_grid, price_obj.maturity)
    current_level = price_obj.payoff(self, event_obj, t_max)

    # Bondary condition
    lower_boundary = price_obj.generate_lb(self, event_obj)
    upper_boundary = price_obj.generate_ub(self, event_obj)

    # Initialize the arrays
    continuation_array = [current_level]
    boundary_array = [current_level]
    solution_array = [current_level]

    # Backward diffusion
    # for t_index in range(self.time_size - 2, -1, -1):
    for t_index in range(t_max - 1, -1, -1):

        # Solve the system to get continuation value
        continuation_value = self.pricing_theta.backward_komogorov(
            model=self,
            price_obj=price_obj,
            event_obj=event_obj,
            t_start=t_max,
            t_index=t_index,
            lower_boundary=lower_boundary,
            upper_boundary=upper_boundary,
            solution_array=current_level,
        )
        # Add the coupon
        continuation_value += price_obj.coupon(event_obj, t_index)

        # Directly use continuation value as the exercise boundary
        exercise_boundary = continuation_value

        # Handle security specific events
        current_level = price_obj.handle_events(
            self,
            event_obj,
            exercise_boundary,
            continuation_value,
            t_index,
        )

        # Keep all steps for debugging purposes
        continuation_array = [continuation_value] + continuation_array
        boundary_array = [exercise_boundary] + boundary_array
```

```python
        solution_array = [current_level] + solution_array

    return continuation_array, boundary_array, solution_array


def forward_diffusion(self, current_level, t_start, t_end, at, bt):
    """
    Step forward from start to end (both inclusive).
    """
    for t_index in range(t_start + 1, t_end + 1):
        current_level = self.calibration_theta.forward_komogorov(
            model=self,
            at=at,
            bt=bt,
            t_start=t_start,
            t_index=t_index,
            solution_array=current_level,
        )

    return current_level


def price_calibration_target(self, dirac, t_start, t_end, at, bt):
    """
    Calculate the price of calibration targets on end date, assuming
    constant at and bt from start to end date.
    """
    # Step from previous to current calibration point
    dirac = self.forward_diffusion(dirac, t_start, t_end, at, bt)

    # ATMF implied volatility
    # strike = self.stock_price * numpy.exp(
    #     (self.riskless_rate - self.dividend_yield) * self.time_grid[t_end]
    # )
    strike = self.stock_forward[t_end]
    call_price = self.dX * numpy.sum(
        dirac * numpy.maximum(self.stock_grid[t_end] - strike, 0)
    )
    call_vol = pricing_utils.bs_cvol(
        price=call_price,
        S=self.stock_price,
        K=strike,
        T=self.time_grid[t_end],
        r=self.r[t_end],
        q=self.dividend_yield,
    )

    # Zero-coupon bond risky spread
    bond_price = self.dX * numpy.sum(dirac)
    bond_spread = -numpy.log(bond_price / self.P[t_end]) / self.time_grid[t_end]

    return call_vol, bond_spread


def joint_root_search(
    self, dirac, t_start, t_end, at, bt, target_vol, target_spread
):
    """
    Simplify two-dimensional root-search problem to inexpensive iteration
    over two one-dimensional root-searches.
    """
    def obj_at(at):
        _, bond_spread = self.price_calibration_target(
            dirac, t_start, t_end, at, bt
        )
        return bond_spread - target_spread


    def obj_bt(bt):
        call_vol, _ = self.price_calibration_target(
            dirac, t_start, t_end, at, bt
        )
        return call_vol - target_vol

    maxiter = 5
    tol = 1e-6
    for numiter in range(1, maxiter + 1):
        # Freeze at and solve for bt
        bt = brentq(obj_bt, 1e-6, 3.0, xtol=tol)

        # Freeze bt and solve for at
        at = brentq(obj_at, 1e-6, 0.5, xtol=tol)

        # Evaluate the targets
        call_vol, bond_spread = self.price_calibration_target(
            dirac, t_start, t_end, at, bt
        )
```

```
            at_err = bond_spread - target_spread
            bt_err = call_vol - target_vol

            # Convergence check
            converged = abs(at_err) < tol and abs(bt_err) < tol
            if converged:
                break

        if not converged:
            raise RuntimeError(f"not converged after maxiter: {maxiter}")

        return at, bt, at_err, bt_err, numiter


    def calibrate(self, raise_error=True):
        """
        Joint calibration of jump intensity and diffusive vol.
        """
        # Initialize arrays
        ats = {}
        bts = {}
        ats_err = {}
        bts_err = {}
        numiters = {}

        # Terminal condition
        index = numpy.searchsorted(self.stock_grid[0], self.stock_price)
        dirac = numpy.zeros(self.space_size)
        dirac[index] = 1.0 / self.dX

        # Bootstrapping
        t_start = 0
        at = self.lam[0]
        bt = self.sigma[0]

        # Better to use this big try except block because otherwise need
        # something like if has_error and raise_error then raise else break
        try:
            for (expiry, target_spread, target_vol) in self.calibration_target:

                t_end = numpy.searchsorted(self.time_grid, expiry)

                at, bt, at_err, bt_err, numiter = self.joint_root_search(
                    dirac, t_start, t_end, at, bt, target_vol, target_spread
                )

                dirac = self.forward_diffusion(dirac, t_start, t_end, at, bt)

                t_start = t_end

                ats[expiry] = at
                bts[expiry] = bt
                ats_err[expiry] = at_err
                bts_err[expiry] = bt_err
                numiters[expiry] = numiter

            # Encapsulate these components
            result = {
                "ats": ats,
                "bts": bts,
                "ats_err": ats_err,
                "bts_err": bts_err,
                "numiters": numiters,
                "error": "",
            }

        except Exception as e:
            message = f"failed to calibrate: {expiry:.2f} as {str(e)}"

            if raise_error:
                raise RuntimeError(message) from e

            result = {
                "ats": ats,
                "bts": bts,
                "ats_err": ats_err,
                "bts_err": bts_err,
                "numiters": numiters,
                "error": message,
            }

        # Return a new instance to avoid potential lingering state issues
        parameters = {
            p.name: getattr(self, p.name)
            for p in inspect.signature(self.__init__).parameters.values()
        }
        if not result["error"]:
```

```
            parameters["default_intensity"] = result["ats"]
            parameters["equity_vol"] = result["bts"]

        model = self.__class__(**parameters)

        return result, model
```

/objects/boundaries.py

```
import numpy


class BoundaryBase(object):
    """
    Base class for all boundary conditions.
    """
    def update_matrix(self, model, t_index, a, b, c):
        raise ValueError("Child must implement")


class LowerDirichlet(BoundaryBase):
    def __init__(self, value_array):
        self.value_array = value_array


    def update_matrix(self, model, t_index, a, b, c):
        D_tl = b[0]
        D_tr = c[0]
        B_t = a[0] * self.value_array[t_index]

        return D_tl, D_tr, B_t


class LowerNeumann(BoundaryBase):
    def __init__(self, order, value):
        self.order = order
        self.value = value


    def update_matrix(self, model, t_index, a, b, c):
        # Zero Delta
        if self.order == 1 and self.value == 0:
            D_tl = b[0]
            D_tr = a[0] + c[0]
            B_t = 0.0

        # Negative One Delta
        elif self.order == 1 and self.value == -1:
            D_tl = b[0]
            D_tr = a[0] + c[0]
            B_t = 2.0 * a[0] * model.dX * numpy.exp(model.X_min + model.dX)

        # Zero Gamma
        elif self.order == 2 and self.value == 0:
            D_tl = b[0] + 4.0 * a[0] / (model.dX + 2.0)
            D_tr = c[0] + a[0] * (model.dX - 2.0) / (model.dX + 2.0)
            B_t = 0.0

        else:
            raise ValueError("Invalid upper boundary")

        return D_tl, D_tr, B_t


class UpperDirichlet(BoundaryBase):
    def __init__(self, value_array):
        self.value_array = value_array


    def update_matrix(self, model, t_index, a, b, c):
        D_bl = a[-1]
        D_br = b[-1]
        B_b = c[-1] * self.value_array[t_index]

        return D_bl, D_br, B_b


class UpperNeumann(BoundaryBase):
    def __init__(self, order, value):
        self.order = order
        self.value = value


    def update_matrix(self, model, t_index, a, b, c):
        # Zero Delta
        if self.order == 1 and self.value == 0:
            D_bl = a[-1] + c[-1]
```

```python
            D_br = b[-1]
            B_b = 0.0

        # One Delta
        elif self.order == 1 and self.value == 1:
            D_bl = a[-1] + c[-1]
            D_br = b[-1]
            B_b = 2.0 * c[-1] * model.dX * numpy.exp(model.X_max - model.dX)

        # Zero Gamma
        elif self.order == 2 and self.value == 0:
            D_bl = a[-1] + c[-1] * (model.dX + 2.0) / (model.dX - 2.0)
            D_br = b[-1] - 4.0 * c[-1] / (model.dX - 2.0)
            B_b = 0.0

        else:
            raise ValueError("Invalid upper boundary")

        return D_bl, D_br, B_b
```

/objects/events.py

```python
import numpy
import pandas
from securities import converts
from utils import parsing_utils


class Event(object):
    """
    Contains all events
    """
    def __init__(self, model, security):

        if hasattr(security, "coupon_schedule"):
            self.coupon_event = CouponEvent(
                model,
                security,
            )

        if hasattr(security, "conversion_schedule"):
            self.conversion_event = ConversionEvent(
                model,
                security,
            )

        if hasattr(security, "call_schedule"):
            self.call_event = CallEvent(
                model,
                security,
            )

        if hasattr(security, "put_schedule"):
            self.put_event = PutEvent(
                model,
                security,
            )


class CouponEvent(object):
    """
    Coupon and accrued.
    """
    def __init__(self, model, security):

        # Parse and fill NA with zeros
        _, _, self.coupon = parsing_utils.parse_schedule(
            model.time_grid,
            security.coupon_schedule,
        )
        self.coupon = pandas.Series(self.coupon).fillna(0).to_numpy()

        # Calculate riskless PV of cashflows
        self.riskless_pv = parsing_utils.pv_future_cashflow(
            model,
            security,
            self.coupon,
        )

        # Need to calculate accrued interests before filling NA
        self.accrued = parsing_utils.accrued_interest(
            model.time_grid,
            security.coupon_schedule,
        )


class ConversionEvent(object):
```

```python
    """
    Issuer call.
    """
    def __init__(self, model, security):
        self.is_convertible, _, self.cr = parsing_utils.parse_schedule(
            model.time_grid,
            security.conversion_schedule,
        )

        self.is_protected, self.hurdle, _ = parsing_utils.parse_schedule(
            model.time_grid,
            security.dividend_protection,
        )

        # Adjust conversion ratio if there is dividend protection
        # TODO: should handle continuous dividend as well
        self.multiplier = numpy.where(
            numpy.logical_and(model.cq > 0, self.is_protected),
            model.stock_price / (model.stock_price - numpy.maximum(model.cq - self.hurdle, 0)),
            1.0,
        )
        self.cr = numpy.cumprod(self.multiplier) * self.cr

        # Fill forward and then backward which will be used for forced
        # conversion when call period is not in conversion schedule
        self.filled_cr = pandas.Series(self.cr).ffill().bfill().to_numpy()


class CallEvent(object):
    """
    Issuer call.
    """
    def __init__(self, model, security):

        # Parse the call schedule which is mixed between
        # hard call and soft calls
        flag_array, trigger_array, price_array = parsing_utils.parse_schedule(
            model.time_grid,
            security.call_schedule,
        )

        # Same size as the stock spline
        soft_callable = []

        for i in range(len(model.time_grid)):
            # Hard call
            if flag_array[i] and numpy.isnan(trigger_array[i]):
                soft_callable.append(
                    numpy.ones_like(model.stock_grid[i], dtype=bool)
                )
            # Soft call
            elif flag_array[i] and not numpy.isnan(trigger_array[i]):
                soft_callable.append(
                    numpy.where(model.stock_grid[i] > trigger_array[i], True, False)
                )
            # Not callable
            else:
                soft_callable.append(
                    numpy.zeros_like(model.stock_grid[i], dtype=bool)
                )

        self.is_callable = flag_array
        self.soft_callable = soft_callable
        self.call_price = price_array

        # Converts specific parameters
        if isinstance(security, converts.ConvertibleBond):
            self.initialize_converts(security)


    def initialize_converts(self, security):
        """
        Only converts have these parameters.
        """
        self.call_notice_period = security.call_notice_period

        # Parameters for call delay modelling
        if security.call_cushion > 0:
            self.delay_call = True
            self.call_speed = (
                -numpy.log(1 - security.call_cushion_prob_1m) / security.call_cushion * 12
            )
        else:
            self.delay_call = False
            self.call_speed = 0.0
```

```
class PutEvent(object):
    """
    Put back to the issuer.
    """
    def __init__(self, model, security):
        # Parse the put schedule which is mixed between
        # hard put and soft puts
        flag_array, trigger_array, price_array = parsing_utils.parse_schedule(
            model.time_grid,
            security.put_schedule,
        )

        # Same size as the stock spline
        soft_puttable = []

        for i in range(len(model.time_grid)):
            # Hard put
            if flag_array[i] and numpy.isnan(trigger_array[i]):
                soft_puttable.append(
                    numpy.ones_like(model.stock_grid[i], dtype=bool)
                )
            # Soft put
            elif flag_array[i] and not numpy.isnan(trigger_array[i]):
                soft_puttable.append(
                    numpy.where(model.stock_grid[i] < trigger_array[i], True, False)
                )
            # Not puttable
            else:
                soft_puttable.append(
                    numpy.zeros_like(model.stock_grid[i], dtype=bool)
                )

        self.is_puttable = flag_array
        self.soft_puttable = soft_puttable
        self.put_price = price_array
```

/scripts/checkout_report.py

```
import pickle
import pandas as pd
from scripts import input_builder


def load_inputs(date, label=""):
    inputs_path = f"C:\\Users\\peter\\Projects\\data\\converts_checkout\\inputs\\{date}"
    if label:
        inputs_path += f"_{label}"
    inputs_path += ".pkl"

    with open(inputs_path, "rb") as f:
        all_inputs = pickle.load(f)

    return all_inputs


def load_outputs(date, label=""):
    outputs_path = f"C:\\Users\\peter\\Projects\\data\\converts_checkout\\outputs\\{date}"
    if label:
        outputs_path += f"_{label}"
    outputs_path += ".parquet"

    all_outputs = pd.read_parquet(outputs_path)

    return all_outputs


def gen_report(date, label=""):
    manifest = input_builder.load_universe(date)
    all_outputs = load_outputs(date, label)
    combined = manifest.merge(all_outputs, on="order_book_id", how="outer")
    combined["stock_price"] = combined["close"]
    combined["parity"] = combined["stock_price"] * combined["conversion_ratio"]
    combined["conversion_premium"] = combined["clean_price"] - combined["parity"]
    combined["edge"] = combined["theo_value"] - combined["clean_price"]
    combined = combined.sort_values("edge")
    combined = combined.reset_index(drop=True)
    combined = combined[[
        "date",
        "order_book_id",
        "stock_code",
        "ts_code",
        "symbol",
        "par_value",
        "redemption_price",
        "maturity_date",
        "TTM",
        "conversion_ratio",
```

```
        "stock_price",
        "realized_252",
        "DI",
        "parity",
        "theo_value",
        "clean_price",
        "dirty_price",
        "trade_type",
        "conversion_premium",
        "edge",
        "default_prob",
        "optional_conv_prob",
        "maturity_conv_prob",
        "call_redemption_prob",
        "forced_conv_prob",
        "put_prob",
        "maturity_redemption_prob",
    ]]
    return combined


def send_report():
    """
    TODO: generate and send email report before release.
    """
    pass
```

/scripts/input_builder.py

```
import sys
import pickle
import numpy as np
import pandas as pd
from scipy.stats import norm


def calc_realized_vol():
    realized_vol = pd.read_parquet(
        "C:\\Users\\peter\\Projects\\data\\converts_checkout\\tushare_adj_stock_price.parquet"
    )
    realized_vol["logreturn"] = np.log(realized_vol["close"] / realized_vol["pre_close"])

    realized_vol = realized_vol.sort_values(["ts_code","trade_date"])
    realized_vol = realized_vol.reset_index(drop=True)
    realized_vol["realized_252"] = realized_vol.groupby("ts_code")["logreturn"].transform(
        lambda x: x.rolling(window=252, min_periods=200).std() * np.sqrt(252)
    )
    realized_vol = realized_vol[["ts_code","trade_date","realized_252"]]
    return realized_vol


def calc_merton_cds(
    acctype,
    market_sharpe=0.8,
    corr_market_asset=0.6,
    cds_recovery=0.4,
    cds_tenor=5,
):
    assert acctype in ["ttm", "mrq"]

    # load financial statements
    financials = pd.read_parquet(
        f"C:\\Users\\peter\\Projects\\data\\converts_checkout\\mikuang_financial_statements_daily.parquet"
    )
    financials["ts_code"] = financials["order_book_id"].apply(lambda x: x.replace(".XSHE", ".SZ").replace(".XSHG", ".SH"))
    # scale to billions
    for col in financials.columns:
        if col not in ["order_book_id", "date", "ts_code"]:
            financials[col] /= 1e9
    financials["non_current_liabilities_ttm_0"] = financials["total_liabilities_ttm_0"] - financials["current_liabilities_ttm_0"]
    financials["non_current_liabilities_mrq_0"] = financials["total_liabilities_mrq_0"] - financials["current_liabilities_mrq_0"]

    # calc equity realized vol
    rlzd_vol = calc_realized_vol()
    rlzd_vol = rlzd_vol.rename(columns={"trade_date": "date", "realized_252": "equity_vol"})

    # merge together
    combined = financials.merge(rlzd_vol, on=["date","ts_code"], how="inner")
    combined["current_ratio"] = combined[f"current_liabilities_{acctype}_0"] / combined[f"total_liabilities_{acctype}_0"]
    combined["avg_current_ratio"] = combined.groupby("date")["current_ratio"].transform(lambda x: x.median())

    # fall back to cross-sectional average ratio if short-term debt or long-term debt is NA
    combined["effective_debt"] = combined[f"current_liabilities_{acctype}_0"] + 0.5 * combined[f"non_current_liabilities_{acctype}_0"]
    combined["effective_debt"] = np.where(
        ~np.isnan(combined["effective_debt"]),
```

```python
        combined["effective_debt"],
        combined[f"total_liabilities_{acctype}_0"] * combined["avg_current_ratio"] + 0.5 * combined[f"total_liabilities_{acctype}_0"] * (1 -
combined["avg_current_ratio"]),
    )
    combined["asset_value"] = combined["market_cap_3"] + combined["effective_debt"]
    combined["de_ratio"] = combined["effective_debt"] / combined["market_cap_3"]
    combined["asset_vol"] = combined["equity_vol"] / (1 + combined["de_ratio"])
    combined["DD"] = np.log(combined["asset_value"] / combined["effective_debt"]) / combined["asset_vol"]
    combined["EDF"] = norm.cdf(-combined["DD"])
    combined["CPD"] = 1 - (1 - combined["EDF"]) ** cds_tenor
    combined["CQD"] = norm.cdf(
        norm.ppf(combined["CPD"]) + market_sharpe * np.sqrt(corr_market_asset ** 2) * np.sqrt(cds_tenor)
    )
    combined["CDS"] = - 1 / cds_tenor * np.log(1 - (1 - cds_recovery) * combined["CQD"])
    combined["CDS"] *= 1e4
    combined["DI"] = np.clip(combined["CDS"] / (1 - cds_recovery), 0, 3000)

    combined = combined.rename(columns={
        f"total_liabilities_{acctype}_0": "book_val_debt",
        "market_cap_3": "market_cap",
    })
    combined = combined[[
        "date",
        "ts_code",
        "market_cap",
        "book_val_debt",
        "effective_debt",
        "de_ratio",
        "equity_vol",
        "asset_vol",
        "DD",
        "CDS",
        "DI",
    ]]
    return combined


def load_universe(date):
    # Take the converts with valid market price
    converts_price = pd.read_parquet(
        "C:\\Users\\peter\\Projects\\data\\converts_checkout\\mikuang_close_price.parquet"
    )
    converts_price = converts_price[converts_price["date"] == date]
    converts_price = converts_price.dropna(subset=["clean_price","dirty_price"])

    # Take out the susepended names because their marks are based on forward fill
    converts_suspends = pd.read_parquet(
        "C:\\Users\\peter\\Projects\\data\\converts_checkout\\mikuang_suspends.parquet"
    )
    converts_price = converts_price.merge(
        converts_suspends, on=["date","order_book_id"], how="left",
    )
    converts_price = converts_price[~(converts_price["is_suspended"] == True)]

    # Remove exchangeables and NA basic terms
    basic_terms = pd.read_parquet(
        "C:\\Users\\peter\\Projects\\data\\converts_checkout\\mikuang_all_instruments.parquet"
    )
    converts_price = converts_price.merge(
        basic_terms, on="order_book_id", how="left",
    )
    converts_price["TTCS"] = np.busday_count(converts_price["date"].values.astype("datetime64[D]"),
        converts_price["conversion_start_date"].values.astype("datetime64[D]")) / 252.0

    converts_price["TTCE"] = np.busday_count(converts_price["date"].values.astype("datetime64[D]"),
        converts_price["conversion_end_date"].values.astype("datetime64[D]")) / 252.0

    converts_price["TTCS"] = np.maximum(converts_price["TTCS"], 0.0)
    converts_price["TTCE"] = np.maximum(converts_price["TTCE"], 0.0)

    converts_price["TTM"] = np.busday_count(converts_price["date"].values.astype("datetime64[D]"),
        converts_price["maturity_date"].values.astype("datetime64[D]")) / 252.0

    converts_price = converts_price[converts_price["bond_type"] == "cb"]
    converts_price = converts_price.dropna(subset=["TTCS","TTCE","TTM","par_value","redemption_price"])

    # Remove NA conversion ratios
    conv_ratio = pd.read_parquet(
        "C:\\Users\\peter\\Projects\\data\\converts_checkout\\mikuang_conversion_prices.parquet"
    )
    conv_ratio = conv_ratio[conv_ratio["effective_date"] < date]
    conv_ratio = conv_ratio.sort_values(["order_book_id", "effective_date"])
    conv_ratio = conv_ratio.groupby("order_book_id").last().reset_index()
    converts_price = converts_price.merge(
        conv_ratio, on="order_book_id", how="left",
    )
    converts_price["conversion_ratio"] = converts_price["par_value"] / converts_price["conversion_price"]
```

```python
    converts_price = converts_price.dropna(subset=["conversion_ratio"])

    # Remove NA stock prices
    stock_price = pd.read_parquet(
        "C:\\Users\\peter\\Projects\\data\\converts_checkout\\tushare_raw_stock_price.parquet"
    )
    stock_price = stock_price[stock_price["trade_date"] == date]
    converts_price["ts_code"] = converts_price["stock_code"].apply(
        lambda x: x.replace(".XSHE", ".SZ").replace(".XSHG", ".SH")
    )
    converts_price = converts_price.merge(
        stock_price, on="ts_code", how="left",
    )
    converts_price = converts_price.dropna(subset=["close"])

    # Remove NA stock vols
    realized_vol = calc_realized_vol()
    realized_vol = realized_vol.rename(columns={"trade_date": "date"})
    realized_vol = realized_vol[realized_vol["date"] == date]
    converts_price = converts_price.merge(
        realized_vol, on=["date", "ts_code"], how="left",
    )
    converts_price = converts_price.dropna(subset=["realized_252"])

    # Add default intensity column
    merton_cds = calc_merton_cds("mrq")
    merton_cds = merton_cds[merton_cds["date"] == date]
    converts_price = converts_price.merge(
        merton_cds, on=["date","ts_code"], how="left",
    )

    converts_price = converts_price.reset_index(drop=True)

    return converts_price


def gen_coupon_dict(date, manifest):
    def _func(group):
        val = group[group["TTC"] > 0].set_index("TTC")["coupon_rate"].to_dict()
        return val
    coupons = pd.read_parquet(
        "C:\\Users\\peter\\Projects\\data\\converts_checkout\\mikuang_coupon_terms.parquet"
    )
    coupons = coupons.merge(
        manifest[["order_book_id", "par_value"]],
        on="order_book_id",
        how="inner",
    )
    coupons["date"] = pd.to_datetime(date)
    coupons["TTC"] = np.busday_count(coupons["date"].values.astype("datetime64[D]"),
            coupons["end_date"].values.astype("datetime64[D]")) / 252.0
    coupons["coupon_rate"] *= coupons["par_value"]
    coupons = coupons.groupby("order_book_id").apply(_func, include_groups=False)
    coupons = coupons.to_dict()
    return coupons


def gen_call_put_dict_given_terms(date, manifest, terms):
    terms["date"] = pd.to_datetime(date)
    terms["TTCS"] = np.busday_count(terms["date"].values.astype("datetime64[D]"),
        terms["start_date"].values.astype("datetime64[D]")) / 252.0
    terms["TTCE"] = np.busday_count(terms["date"].values.astype("datetime64[D]"),
        terms["end_date"].values.astype("datetime64[D]")) / 252.0

    terms["TTCS"] = np.maximum(terms["TTCS"], 0.0)
    terms["TTCE"] = np.maximum(terms["TTCE"], 0.0)

    d = {}
    for _, row in manifest.iterrows():
        call = terms[terms["order_book_id"] == row["order_book_id"]]

        dd = {}
        if len(call) > 0:
            call = call.drop_duplicates(["start_date","end_date"])
            for _, sched in call.iterrows():
                dd[(sched["TTCS"], sched["TTCE"])] = (sched["level"] * row["conversion_price"], row["par_value"])
        d[row["order_book_id"]] = dd

    return d


def gen_call_dict(date, manifest):
    call_terms = pd.read_parquet(
        "C:\\Users\\peter\\Projects\\data\\converts_checkout\\mikuang_call_terms.parquet"
    )

    return gen_call_put_dict_given_terms(date, manifest, call_terms)
```

```python
def gen_put_dict(date, manifest):
    put_terms = pd.read_parquet(
        "C:\\Users\\peter\\Projects\\data\\converts_checkout\\mikuang_put_terms.parquet"
    )

    return gen_call_put_dict_given_terms(date, manifest, put_terms)


def gen_div_dict(date, manifest):
    divs = pd.read_parquet("C://Users//peter//Projects//data//converts_checkout/mikuang_dividends.parquet")
    divs["div per share"] = divs["dividend_cash_before_tax"] / divs["round_lot"]
    divs["date"] = pd.to_datetime(date)
    divs["ex_dividend_date"] = pd.to_datetime(divs["ex_dividend_date"])
    divs["TTD"] = np.busday_count(divs["date"].values.astype("datetime64[D]"),
            divs["ex_dividend_date"].values.astype("datetime64[D]")) / 252.0
    divs = divs[(divs["TTD"] > -1) & (divs["TTD"] < 0)]
    divs = divs.groupby("order_book_id")["div per share"].agg(["count","sum"])
    divs = divs.reset_index(drop=False)
    divs["interval"] = 1.0 / divs["count"]
    divs["amount"] = divs["sum"] / divs["count"]

    d = {}
    for _, row in manifest.iterrows():
        div = divs[divs["order_book_id"] == row["stock_code"]]

        dd = {}
        if len(div) > 0:
            interval = div["interval"].values[0]
            amount = div["amount"].values[0]

            t = interval
            while t < row["TTM"]:
                dd[t] = amount
                t += interval

        d[row["order_book_id"]] = dd
    return d


def load_yield_curve(date):
    treasury = pd.read_parquet(
        "C:\\Users\\peter\\Projects\\data\\converts_checkout\\mikuang_yield_curve.parquet"
    )
    treasury = treasury[treasury["trading_date"] == date]
    treasury = pd.melt(treasury, id_vars="trading_date")
    treasury["term"] = treasury["variable"].map({
        "1M": 1.0 / 12.0,
        "3M": 3.0 / 12.0,
        "6M": 6.0 / 12.0,
        "9M": 9.0 / 12.0,
        "1Y": 1.0,
        "2Y": 2.0,
        "3Y": 3.0,
        "5Y": 5.0,
        "10Y": 10.0,
    })
    assert len(treasury.dropna(subset=["term"])) == 9
    treasury = treasury.sort_values("term")
    treasury = {
        "term": treasury["term"].tolist(),
        "yield": treasury["value"].tolist(),
    }
    return treasury


def build_inputs(date, label=""):
    """
    Create input objects.
    """
    manifest = load_universe(date)
    yield_curve = load_yield_curve(date)
    coupon_dict = gen_coupon_dict(date, manifest)
    call_dict = gen_call_dict(date, manifest)
    put_dict = gen_put_dict(date, manifest)
    div_dict = gen_div_dict(date, manifest)

    all_inputs = {}
    for _, row in manifest.iterrows():
        all_inputs[row["order_book_id"]] = {
            "spline_size": 50,
            "time_size": min(max(int(row["TTM"] * 300), 500), 3000),
            "num_paths": 1000,
            "random_seed": 42,
            "antithetic_variable": False,
            "regression_method": "all",
```

```python
            "space_size": 1000,
            "min_space": 120,
            "max_space": 120,
            "calibration_target": [],
            "calibration_scheme": "rannacher",
            "pricing_scheme": "rannacher",
            "pde_solver": "thomas",
            "max_years": row["TTM"],
            "riskless_rate": np.interp(row["TTM"], yield_curve["term"], yield_curve["yield"]),
            # "riskless_rate": dict(zip(yield_curve["term"], yield_curve["yield"])),
            "log_discount_func": "cubic",
            "stock_price": row["close"],
            "equity_vol": row["realized_252"],
            "dividend_yield": 0.0,
            # "cash_dividend": div_dict[row["order_book_id"]],
            "cash_dividend": {},
            # "dividend_protection": {(0, row["TTM"]): (0, np.nan)},
            "dividend_protection": {},
            "default_intensity": row["DI"] / 1e4 if not np.isnan(row["DI"]) else 0.0,
            "equity_to_credit": 0.5,
            "maturity": row["TTM"],
            "face_value": row["redemption_price"],
            "coupon_schedule": coupon_dict[row["order_book_id"]],
            "recovery_rate": 0.2,
            "recovery_type": "par",
            "conversion_schedule": {(row["TTCS"], row["TTCE"]): row["conversion_ratio"]},
            "call_schedule": call_dict[row["order_book_id"]],
            "call_notice_period": 20,
            "call_cushion": 0.4,
            "call_cushion_prob_1m": 0.5,
            "put_schedule": put_dict[row["order_book_id"]],
        }

    inputs_path = f"C:\\Users\\peter\\Projects\\data\\converts_checkout\\inputs\\{date}"
    if label:
        inputs_path += f"_{label}"
    inputs_path += ".pkl"

    with open(inputs_path, "wb") as f:
        pickle.dump(all_inputs, f)

    print(f"{date}: saved {len(all_inputs)} inputs!")
```

/scripts/master.py

```python
##########################################################################################

# Navigate to inside pricer folder and run the below
# python -m scripts.master [date] [model type] [label]

# This is for convenience purpose only for now, later on
# each of these child tasks should become independent tasks

##########################################################################################

import sys
import time
from scripts import input_builder
from scripts import sequencer


def run(args):
    if len(args) == 3:
        date = args[1]
        model_type = args[2]
        label = ""
    elif len(args) == 4:
        date = args[1]
        model_type = args[2]
        label = args[3]
    else:
        sys.exit(1)

    assert model_type in ["crr", "lsmc", "pde"]

    print(f"Date={date}, Model={model_type}, Label={label}")

    start_time = time.time()
    input_builder.build_inputs(date, label)
    sequencer.valuate_inputs(date, model_type, label)

    end_time = time.time()
    elapsed_time = end_time - start_time
    minutes, seconds = int(elapsed_time // 60), int(elapsed_time % 60)
    print(f"Execution time: {minutes} minute(s) and {seconds} second(s)")
```

```
if __name__ == "__main__":
    run(sys.argv)
```

/scripts/sequencer.py

```
import pickle
import numpy as np
import pandas as pd
from calculation import calculator


def valuate_inputs(date, model_type, label=""):
    """
    Transform input objects into output objects.
    """
    inputs_path = f"C:\\Users\\peter\\Projects\\data\\converts_checkout\\inputs\\{date}"
    if label:
        inputs_path += f"_{label}"
    inputs_path += ".pkl"

    with open(inputs_path, "rb") as f:
        all_inputs = pickle.load(f)

    all_outputs = []
    for order_book_id, input in all_inputs.items():
        model, _, output = calculator.valuate(input, model_type)

        if model_type == "crr":
            all_outputs.append({
                "order_book_id": order_book_id,
                "theo_value": np.interp(input["stock_price"], model.stock_grid[0], output["price"][0]),
                "default_prob": np.interp(input["stock_price"], model.stock_grid[0], output["default_prob"][0]),
                "optional_conv_prob": np.interp(input["stock_price"], model.stock_grid[0], output["optional_conv_prob"][0]),
                "maturity_conv_prob": np.interp(input["stock_price"], model.stock_grid[0], output["maturity_conv_prob"][0]),
                "call_redemption_prob": np.interp(input["stock_price"], model.stock_grid[0], output["call_redemption_prob"][0]),
                "forced_conv_prob": np.interp(input["stock_price"], model.stock_grid[0], output["forced_conv_prob"][0]),
                "put_prob": np.interp(input["stock_price"], model.stock_grid[0], output["put_prob"][0]),
                "maturity_redemption_prob": np.interp(input["stock_price"], model.stock_grid[0], output["maturity_redemption_prob"][0]),
            })
        elif model_type == "lsmc":
            all_outputs.append({
                "order_book_id": order_book_id,
                "theo_value": output["price"][0].mean(),
                "default_prob": output["default_prob"][0].mean(),
                "optional_conv_prob": output["optional_conv_prob"][0].mean(),
                "maturity_conv_prob": output["maturity_conv_prob"][0].mean(),
                "call_redemption_prob": output["call_redemption_prob"][0].mean(),
                "forced_conv_prob": output["forced_conv_prob"][0].mean(),
                "put_prob": output["put_prob"][0].mean(),
                "maturity_redemption_prob": output["maturity_redemption_prob"][0].mean(),
            })
        elif model_type == "pde":
            all_outputs.append({
                "order_book_id": order_book_id,
                "theo_value": np.interp(input["stock_price"], model.stock_grid[0], output["price"][0]),
                "default_prob": np.interp(input["stock_price"], model.stock_grid[0], output["default_prob"][0]),
                "optional_conv_prob": np.interp(input["stock_price"], model.stock_grid[0], output["optional_conv_prob"][0]),
                "maturity_conv_prob": np.interp(input["stock_price"], model.stock_grid[0], output["maturity_conv_prob"][0]),
                "call_redemption_prob": np.interp(input["stock_price"], model.stock_grid[0], output["call_redemption_prob"][0]),
                "forced_conv_prob": np.interp(input["stock_price"], model.stock_grid[0], output["forced_conv_prob"][0]),
                "put_prob": np.interp(input["stock_price"], model.stock_grid[0], output["put_prob"][0]),
                "maturity_redemption_prob": np.interp(input["stock_price"], model.stock_grid[0], output["maturity_redemption_prob"][0]),
            })

        print(f"[{date}] [{len(all_outputs)}|{len(all_inputs)}]: finished pricing {order_book_id} ..")

    all_outputs = pd.DataFrame(all_outputs)
    outputs_path = f"C:\\Users\\peter\\Projects\\data\\converts_checkout\\outputs\\{date}"
    if label:
        outputs_path += f"_{label}"
    outputs_path += ".parquet"

    all_outputs.to_parquet(outputs_path)
    print(f"{date}: saved {len(all_outputs)} model outputs!")
```

/securities/analytics.py

```
import numpy
from objects import boundaries


class AnalyticsBase(object):
    """
```

```python
    Wrapper on a security object.

    Will point to the method in the security object is not overriden.
    """
    def __init__(self, security, result_dict):
        self.maturity = security.maturity
        self.security = security
        self.result_dict = result_dict


class Price(AnalyticsBase):
    """
    Theoretical price.
    """
    @classmethod
    def to_string(cls):
        return "price"

    def __getattr__(self, name):
        return getattr(self.security, name)


class Probability(AnalyticsBase):
    """
    Default, conversion, call and put probabilities.
    """
    def discount(self, model, t_index):
        return 0.0


    def coupon(self, event_obj, t_index):
        return 0.0


    def estimate_exercise_boundary(
        self,
        model,
        event_obj,
        continuation_value,
        t_index,
    ):
        return continuation_value


    def handle_events(
        self,
        model,
        event_obj,
        exercise_boundary,
        continuation_value,
        t_index,
    ):
        # Events
        coupon_event = event_obj.coupon_event
        conversion_event = event_obj.conversion_event
        call_event = event_obj.call_event
        put_event = event_obj.put_event

        # Initialized the array
        probability = continuation_value
        theo_boundary = self.result_dict["price"]["boundary"][t_index]
        theo_value = self.result_dict["price"]["continuation"][t_index]

        # Handle optional conversion
        if conversion_event.is_convertible[t_index]:
            conversion_value = (
                conversion_event.cr[t_index] * model.stock_grid[t_index]
                + coupon_event.coupon[t_index]
                + coupon_event.accrued[t_index]
            )
            probability = self.probability_after_conversion(
                conversion_value,
                theo_boundary,
                probability,
            )
            theo_value = numpy.where(
                conversion_value > theo_boundary,
                conversion_value,
                theo_value,
            )

        # Handle call
        if call_event.is_callable[t_index]:
            # The holder receives call price if choose redemption
            # No screw clause is more popular in the US
            redemption_value = (
                numpy.full_like(model.stock_grid[t_index], call_event.call_price[t_index])
```

```
                + coupon_event.coupon[t_index]
                + coupon_event.accrued[t_index]
            )

            # The holder receives coupon and accrued if choose conversion
            forced_conversion_value = (
                conversion_event.filled_cr[t_index] * model.stock_grid[t_index]
                + coupon_event.coupon[t_index]
                + coupon_event.accrued[t_index]
            )

            prob_call = self.security.issuer_call_probability(
                model,
                event_obj,
                redemption_value,
                self.result_dict["price"]["boundary"][t_index],
                t_index,
            )

            probability = self.probability_after_call(
                prob_call,
                redemption_value,
                forced_conversion_value,
                probability,
            )

            theo_call = self.security.handle_issuer_call(
                model,
                event_obj,
                redemption_value,
                forced_conversion_value,
                t_index,
            )

            theo_value = theo_call * prob_call + theo_value * (1 - prob_call)

        # Handle put
        if put_event.is_puttable[t_index]:

            put_price = put_event.put_price[t_index]

            probability = self.probability_after_put(
                put_price,
                theo_value,
                probability,
            )

            theo_value = numpy.where(
                put_event.soft_puttable[t_index],
                numpy.maximum(put_price, theo_value),
                theo_value,
            )

        return probability


class DefaultProbability(Probability):

    def payoff(self, model, event_obj, t_max):
        return numpy.zeros_like(model.stock_grid[t_max])


    def recovery(self, model, event_obj, t_index):
        return 1.0


    @classmethod
    def to_string(cls):
        return "default_prob"


    def probability_after_conversion(
        self,
        conversion_value,
        theo_value,
        probability,
    ):
        return numpy.where(conversion_value - theo_value > 1e-3, 0.0, probability)


    def probability_after_call(
        self,
        prob_call,
        redemption_value,
        forced_conversion_value,
        probability,
    ):
```

```python
        return 0.0 * prob_call + probability * (1 - prob_call)


    def probability_after_put(
        self,
        put_price,
        theo_value,
        probability,
    ):
        return numpy.where(put_price - theo_value > 1e-3, 0.0, probability)


    def generate_lb(self, model, event_obj):
        return boundaries.LowerNeumann(order=1, value=0)


    def generate_ub(self, model, event_obj):
        return boundaries.UpperNeumann(order=1, value=0)


class OptionalConvProbability(Probability):

    def payoff(self, model, event_obj, t_max):
        return numpy.zeros_like(model.stock_grid[t_max])


    def recovery(self, model, event_obj, t_index):
        return 0.0


    @classmethod
    def to_string(cls):
        return "optional_conv_prob"


    def probability_after_conversion(
        self,
        conversion_value,
        theo_value,
        probability,
    ):
        return numpy.where(conversion_value - theo_value > 1e-3, 1.0, probability)


    def probability_after_call(
        self,
        prob_call,
        redemption_value,
        forced_conversion_value,
        probability,
    ):
        return 0.0 * prob_call + probability * (1 - prob_call)


    def probability_after_put(
        self,
        put_price,
        theo_value,
        probability,
    ):
        return numpy.where(put_price - theo_value > 1e-3, 0.0, probability)


    def generate_lb(self, model, event_obj):
        value_array = numpy.zeros_like(model.time_grid)
        return boundaries.LowerDirichlet(value_array)


    def generate_ub(self, model, event_obj):
        return boundaries.UpperNeumann(order=1, value=0)


class MaturityConvProbability(Probability):

    def payoff(self, model, event_obj, t_max):
        # Events
        conversion_event = event_obj.conversion_event

        if conversion_event.is_convertible[t_max]:
            return numpy.where(
                conversion_event.cr[t_max] * model.stock_grid[t_max] > self.security.face_value,
                1.0,
                0.0,
            )

        return numpy.zeros_like(model.stock_grid[t_max])
```

```
    def recovery(self, model, event_obj, t_index):
        return 0.0


    @classmethod
    def to_string(cls):
        return "maturity_conv_prob"


    def probability_after_conversion(
        self,
        conversion_value,
        theo_value,
        probability,
    ):
        return numpy.where(conversion_value - theo_value > 1e-3, 1.0, probability)


    def probability_after_conversion(
        self,
        conversion_value,
        theo_value,
        probability,
    ):
        return numpy.where(conversion_value - theo_value > 1e-3, 0.0, probability)


    def probability_after_call(
        self,
        prob_call,
        redemption_value,
        forced_conversion_value,
        probability,
    ):
        return 0.0 * prob_call + probability * (1 - prob_call)


    def probability_after_put(
        self,
        put_price,
        theo_value,
        probability,
    ):
        return numpy.where(
            put_price - theo_value > 1e-3, 0.0, probability,
        )


    def generate_lb(self, model, event_obj):
        value_array = numpy.zeros_like(model.time_grid)
        return boundaries.LowerDirichlet(value_array)


    def generate_ub(self, model, event_obj):
        return boundaries.UpperNeumann(order=1, value=0)


class CallRedemptionProbability(Probability):

    def payoff(self, model, event_obj, t_max):
        return numpy.zeros_like(model.stock_grid[t_max])


    def recovery(self, model, event_obj, t_index):
        return 0.0


    @classmethod
    def to_string(cls):
        return "call_redemption_prob"


    def probability_after_conversion(
        self,
        conversion_value,
        theo_value,
        probability,
    ):
        return numpy.where(conversion_value - theo_value > 1e-3, 0.0, probability)


    def probability_after_call(
        self,
        prob_call,
        redemption_value,
        forced_conversion_value,
```

```python
        probability,
    ):
        return (
            numpy.where(redemption_value > forced_conversion_value, 1.0, 0.0) * prob_call
            + probability * (1 - prob_call)
        )


    def probability_after_put(
        self,
        put_price,
        theo_value,
        probability,
    ):
        return numpy.where(put_price - theo_value > 1e-3, 0.0, probability)


    def generate_lb(self, model, event_obj):
        value_array = numpy.zeros_like(model.time_grid)
        return boundaries.LowerDirichlet(value_array)


    def generate_ub(self, model, event_obj):
        return boundaries.UpperNeumann(order=1, value=0)


class ForcedConvProbability(Probability):

    def payoff(self, model, event_obj, t_max):
        return numpy.zeros_like(model.stock_grid[t_max])


    def recovery(self, model, event_obj, t_index):
        return 0.0


    @classmethod
    def to_string(cls):
        return "forced_conv_prob"


    def probability_after_conversion(
        self,
        conversion_value,
        theo_value,
        probability,
    ):
        return numpy.where(conversion_value - theo_value > 1e-3, 0.0, probability)


    def probability_after_call(
        self,
        prob_call,
        redemption_value,
        forced_conversion_value,
        probability,
    ):
        return (
            numpy.where(redemption_value > forced_conversion_value, 0.0, 1.0) * prob_call
            + probability * (1 - prob_call)
        )


    def probability_after_put(
        self,
        put_price,
        theo_value,
        probability,
    ):
        return numpy.where(put_price - theo_value > 1e-3, 0.0, probability)


    def generate_lb(self, model, event_obj):
        value_array = numpy.zeros_like(model.time_grid)
        return boundaries.LowerDirichlet(value_array)


    def generate_ub(self, model, event_obj):
        return boundaries.UpperNeumann(order=1, value=0)


class PutProbability(Probability):

    def payoff(self, model, event_obj, t_max):
        return numpy.zeros_like(model.stock_grid[t_max])
```

```
    def recovery(self, model, event_obj, t_index):
        return 0.0


    @classmethod
    def to_string(cls):
        return "put_prob"


    def probability_after_conversion(
        self,
        conversion_value,
        theo_value,
        probability,
    ):
        return numpy.where(conversion_value - theo_value > 1e-3, 0.0, probability)


    def probability_after_call(
        self,
        prob_call,
        redemption_value,
        forced_conversion_value,
        probability,
    ):
        return 0.0 * prob_call + probability * (1 - prob_call)


    def probability_after_put(
        self,
        put_price,
        theo_value,
        probability,
    ):
        return numpy.where(put_price - theo_value > 1e-3, 1.0, probability)


    def generate_lb(self, model, event_obj):
        return boundaries.LowerNeumann(order=1, value=0)


    def generate_ub(self, model, event_obj):
        value_array = numpy.zeros_like(model.time_grid)
        return boundaries.UpperDirichlet(value_array)


class MaturityRedemptionProbability(Probability):

    def payoff(self, model, event_obj, t_max):
        # Events
        conversion_event = event_obj.conversion_event

        if conversion_event.is_convertible[t_max]:
            return numpy.where(
                conversion_event.cr[t_max] * model.stock_grid[t_max] > self.security.face_value,
                0.0,
                1.0,
            )

        return numpy.ones_like(model.stock_grid[t_max])


    def recovery(self, model, event_obj, t_index):
        return 0.0


    @classmethod
    def to_string(cls):
        return "maturity_redemption_prob"


    def probability_after_conversion(
        self,
        conversion_value,
        theo_value,
        probability,
    ):
        return numpy.where(conversion_value - theo_value > 1e-3, 0.0, probability)


    def probability_after_call(
        self,
        prob_call,
        redemption_value,
        forced_conversion_value,
        probability,
    ):
```

```
        return 0.0 * prob_call + probability * (1 - prob_call)


    def probability_after_put(
        self,
        put_price,
        theo_value,
        probability,
    ):
        return numpy.where(put_price - theo_value > 1e-3, 0.0, probability)


    def generate_lb(self, model, event_obj):
        return boundaries.LowerNeumann(order=1, value=0)


    def generate_ub(self, model, event_obj):
        value_array = numpy.zeros_like(model.time_grid)
        return boundaries.UpperDirichlet(value_array)
```

/securities/bonds.py

```
import numpy
from objects import boundaries
from securities import security_base


class Bond(security_base.SecurityBase):
    def __init__(
        self,
        maturity,
        face_value,
        coupon_schedule,
        call_schedule,
        put_schedule,
        recovery_rate,
        recovery_type,
    ):
        """
        Defaultable bond.
        """
        self.maturity = maturity
        self.face_value = face_value
        self.coupon_schedule = coupon_schedule
        self.call_schedule = call_schedule
        self.put_schedule = put_schedule
        self.recovery_rate = recovery_rate
        self.recovery_type = recovery_type


    def discount(self, model, t_index):
        """
        Riskless discount factor during dt.
        """
        return model.r[t_index]


    def coupon(self, event_obj, t_index):
        """
        PV of coupon from [start_t, end_t) at start_t.
        """
        return event_obj.coupon_event.coupon[t_index]


    def payoff(self, model, event_obj, t_max):
        """
        Terminal payoff including principal and coupon.
        """
        principal = numpy.ones_like(model.stock_grid[t_max]) * self.face_value
        coupon = event_obj.coupon_event.coupon[t_max]

        return principal + coupon


    def recovery(self, model, event_obj, t_index):
        """
        Assumes par recovery.
        """
        if self.recovery_type == "par":
            recovery_value = self.face_value * self.recovery_rate
        elif self.recovery_type == "pv":
            recovery_value = event_obj.coupon_event.riskless_pv[t_index + 1] * self.recovery_rate
        else:
            raise ValueError(f"Invalid type: {self.recovery_type}")

        return recovery_value
```

```
def get_riskfree_bond_floor(self, model, event_obj):
    """
    Each entry is the bond floor at that time.
    """
    # Events
    coupon_event = event_obj.coupon_event

    # Initialize array
    bond_floor = numpy.zeros_like(model.time_grid)

    # Start from maturity date and move backwards
    t_max = numpy.searchsorted(model.time_grid, self.maturity)
    bond_floor[t_max] = self.face_value + coupon_event.coupon[t_max]

    # for t_index in range(model.time_size - 2, -1, -1):
    for t_index in range(t_max - 1, -1, -1):
        bond_floor[t_index] = (
            numpy.exp(-model.r[t_index] * model.dt)
            * bond_floor[t_index + 1]
            + coupon_event.coupon[t_index]
        )
        # Handle call and put but need to make sure it does not
        # go inside the function overriden by subclass even if
        # the function called from subclass
        bond_floor[t_index] = Bond.handle_events(
            self,
            model,
            event_obj,
            bond_floor[t_index],
            bond_floor[t_index],
            t_index,
        )
    return bond_floor


def get_risky_bond_floor(self, model, event_obj):
    """
    Each entry is the bond floor at that time.
    """
    # Events
    coupon_event = event_obj.coupon_event

    # Initialize array
    bond_floor = numpy.zeros_like(model.time_grid)

    # Start from maturity date and move backwards
    t_max = numpy.searchsorted(model.time_grid, self.maturity)
    bond_floor[t_max] = self.face_value + coupon_event.coupon[t_max]

    # for t_index in range(model.time_size - 2, -1, -1):
    for t_index in range(t_max - 1, -1, -1):
        prob_default = 1 - numpy.exp(-model.lam[t_index] * model.dt)
        bond_floor[t_index] = (
            numpy.exp(-model.r[t_index] * model.dt)
            * (1 - prob_default)
            * bond_floor[t_index + 1]

            + numpy.exp(-model.r[t_index] * model.dt)
            * prob_default
            * self.recovery(self, event_obj, t_index)

            + coupon_event.coupon[t_index]
        )
        # Handle call and put but need to make sure it does not
        # go inside the function overriden by subclass even if
        # the function called from subclass
        bond_floor[t_index] = Bond.handle_events(
            self,
            model,
            event_obj,
            bond_floor[t_index],
            bond_floor[t_index],
            t_index,
        )

    return bond_floor


def handle_events(
    self,
    model,
    event_obj,
    exercise_boundary,
    continuation_value,
    t_index,
):
```

```
        # Events
        coupon_event = event_obj.coupon_event
        call_event = event_obj.call_event
        put_event = event_obj.put_event

        # Initialized the array
        theo_value = continuation_value

        # Handle call
        if call_event.is_callable[t_index]:

            redemption_value = (
                call_event.call_price[t_index]
                + coupon_event.coupon[t_index]
                + coupon_event.accrued[t_index]
            )

            theo_value = numpy.where(
                exercise_boundary > redemption_value,
                redemption_value,
                continuation_value,
            )

        # Handle put
        if put_event.is_puttable[t_index]:

            put_price = put_event.put_price[t_index]

            theo_value = numpy.maximum(put_price, theo_value)

        return theo_value


    def generate_lb(self, model, event_obj):
        value_array = [
            self.recovery(model, event_obj, t_index)
            for t_index in range(len(model.time_grid))
        ]
        return boundaries.LowerDirichlet(value_array)


    def generate_ub(self, model, event_obj):
        return boundaries.UpperNeumann(order=1, value=0)
```

/securities/converts.py

```
import numpy
from objects import boundaries
from securities import bonds
from utils import pricing_utils
import statsmodels.api as sm


class ConvertibleBond(bonds.Bond):
    def __init__(
        self,
        maturity,
        face_value,
        coupon_schedule,
        recovery_rate,
        recovery_type,
        conversion_schedule,
        call_schedule,
        call_notice_period,
        call_cushion,
        call_cushion_prob_1m,
        put_schedule,
        dividend_protection,
    ):
        """
        Convertible bond.
        """
        super().__init__(
            maturity=maturity,
            face_value=face_value,
            coupon_schedule=coupon_schedule,
            call_schedule=call_schedule,
            put_schedule=put_schedule,
            recovery_rate=recovery_rate,
            recovery_type=recovery_type,
        )
        self.conversion_schedule = conversion_schedule
        self.call_notice_period = call_notice_period
        self.call_cushion = call_cushion
        self.call_cushion_prob_1m = call_cushion_prob_1m
        self.dividend_protection = dividend_protection
```

```
def payoff(self, model, event_obj, t_max):
    """
    Terminal payoff.
    """
    # Events
    coupon_event = event_obj.coupon_event
    conversion_event = event_obj.conversion_event

    # Principal redemption and final coupon
    redemption_value = (
        numpy.full_like(model.stock_grid[t_max], self.face_value)
        + coupon_event.coupon[t_max]
    )

    # Conversion value
    if conversion_event.is_convertible[t_max]:
        return numpy.maximum(
            redemption_value,
            conversion_event.cr[t_max] * model.stock_grid[t_max] + coupon_event.coupon[t_max]
        )

    return redemption_value


def issuer_call_probability(
    self,
    model,
    event_obj,
    redemption_value,
    continuation_value,
    t,
):
    """
    Simple call probability based on Duffie 2012 (see Handbook p200).

    Can further add features such as volatility, dividend and so on.
    """
    call_event = event_obj.call_event

    # Empirically issuers delay call until a safety cushion is reached
    if call_event.delay_call:

        # Cap the ratio to avoid overflow inside exp()
        excess_ratio = numpy.clip(
            continuation_value / redemption_value - 1,
            0.0,
            700.0,
        )

        prob_call = numpy.where(
            continuation_value - redemption_value > 1e-3,
            1 - numpy.exp(-call_event.call_speed * excess_ratio * model.dt),
            0.0,
        )

    else:
        prob_call = numpy.where(
            continuation_value - redemption_value > 1e-3, 1.0, 0.0
        )

    # Override to zero if the region is not soft-callable
    prob_call = numpy.where(
        call_event.soft_callable[t],
        prob_call,
        0.0,
    )

    return prob_call


def handle_issuer_call(
    self,
    model,
    event_obj,
    redemption_value,
    forced_conversion_value,
    t_index,
):
    """
    See Eq 6.117 payoff is max(CR * S, K) = CR * S + CR * max(0, K / CR - S).

    More generalized form is max(CR * S + a, K + b) which can be rewritten as

    CR * S + a + CR * max(0, (K + b - a) / CR - S).
```

```python
    Keep it simple and assume the option is European for now.
    """
    # Events
    call_event = event_obj.call_event
    conversion_event = event_obj.conversion_event

    if call_event.call_notice_period == 0:

        return numpy.maximum(redemption_value, forced_conversion_value)

    else:

        option_strike = call_event.call_price[t_index] / conversion_event.filled_cr[t_index]

        expiry = call_event.call_notice_period / 360.0

        # FIXME: should use forward rate from t_i to T
        # Currently this is using forward rate from t_i to t_{i+1}
        interest_rate = model.r[t_index]

        option_value = conversion_event.filled_cr[t_index] * pricing_utils.bs_put(
            S=model.stock_grid[t_index],
            K=option_strike,
            T=expiry,
            r=interest_rate,
            q=model.dividend_yield,
            l=model.lam[t_index],
            sigma=model.sigma[t_index],
        )

        theo_call = forced_conversion_value + option_value

        return theo_call


def estimate_exercise_boundary(
    self,
    model,
    event_obj,
    continuation_value,
    t_index,
):
    """
    Estimate the continuation value for LSMC.
    """
    # Events
    conversion_event = event_obj.conversion_event
    call_event = event_obj.call_event
    put_event = event_obj.put_event

    # Skip if there is no event to handle
    if not (
        conversion_event.is_convertible[t_index]
        or call_event.is_callable[t_index]
        or put_event.is_puttable[t_index]
    ):
        return continuation_value

    # Initialize the array
    estimate = numpy.array(continuation_value)

    # Determine which samples are included in the regression
    if model.regression_method == "all":
        flag = numpy.ones_like(continuation_value, dtype=bool)

    elif model.regression_method == "itm":
        flag = model.stock_grid[t_index] > self.face_value / conversion_event.cr[t_index]

    else:
        raise ValueError(f"Invalid method: {model.regression_method}")

    # Override the continuation value of these samples
    # This is faster than statsmodel API
    if flag.any():
        basis = numpy.stack([
            numpy.ones_like(model.stock_grid[t_index][flag]),
            model.stock_grid[t_index][flag],
            model.stock_grid[t_index][flag] ** 2,
            model.stock_grid[t_index][flag] ** 3,
        ], axis=1)

        result = sm.OLS(continuation_value[flag], basis).fit()
        estimate[flag] = result.fittedvalues

    return estimate
```

```python
def handle_events(
    self,
    model,
    event_obj,
    exercise_boundary,
    continuation_value,
    t_index,
):
    """
    Conversion, call and put.
    """
    # Events
    coupon_event = event_obj.coupon_event
    conversion_event = event_obj.conversion_event
    call_event = event_obj.call_event
    put_event = event_obj.put_event

    # Initialized the array
    theo_value = continuation_value

    # Handle optional conversion
    if conversion_event.is_convertible[t_index]:
        conversion_value = (
            conversion_event.cr[t_index] * model.stock_grid[t_index]
            + coupon_event.coupon[t_index]
            + coupon_event.accrued[t_index]
        )
        theo_value = numpy.where(
            conversion_value > exercise_boundary,
            conversion_value,
            theo_value,
        )

    # Handle call
    if call_event.is_callable[t_index]:
        # The holder receives call price if choose redemption together with
        # coupon if the day is a coupon date (no screw clause is more popular in US)
        # as well as the accrued interests
        redemption_value = (
            numpy.full_like(model.stock_grid[t_index], call_event.call_price[t_index])
            + coupon_event.coupon[t_index]
            + coupon_event.accrued[t_index]
        )

        # The holder receives coupon and accrued if choose conversion
        forced_conversion_value = (
            conversion_event.filled_cr[t_index] * model.stock_grid[t_index]
            + coupon_event.coupon[t_index]
            + coupon_event.accrued[t_index]
        )

        prob_call = self.issuer_call_probability(
            model,
            event_obj,
            redemption_value,
            exercise_boundary,
            t_index,
        )

        theo_call = self.handle_issuer_call(
            model,
            event_obj,
            redemption_value,
            forced_conversion_value,
            t_index,
        )

        theo_value = theo_call * prob_call + theo_value * (1 - prob_call)

    # Handle put
    if put_event.is_puttable[t_index]:

        put_price = put_event.put_price[t_index]

        theo_value = numpy.where(
            put_event.soft_puttable[t_index],
            numpy.maximum(put_price, theo_value),
            theo_value,
        )

    return theo_value


def generate_ub(self, model, event_obj):
    return boundaries.UpperNeumann(order=1, value=1)
```

```
    def to_straight_bond(self):
        """
        Cast to a straight bond without call or put.
        """
        return bonds.Bond(
            maturity=self.maturity,
            face_value=self.face_value,
            coupon_schedule=self.coupon_schedule,
            recovery_rate=self.recovery_rate,
        )
```

/securities/forwards.py

```
import numpy
from objects import boundaries
from securities import security_base


class Forward(security_base.SecurityBase):
    def __init__(self, strike, maturity):
        self.strike = strike
        self.maturity = maturity


    def discount(self, model, t_index):
        return model.r[t_index]


    def coupon(self, event_obj, t_index):
        return 0.0


    def payoff(self, model, event_obj, t_max):
        return model.stock_grid[t_max] - self.strike


    def recovery(self, model, event_obj, t_index):
        return 0.0


    def estimate_exercise_boundary(
        self,
        model,
        event_obj,
        continuation_value,
        t_index,
    ):
        """
        Estimate the continuation value for LSMC.
        """
        raise NotImplementedError


    def handle_events(
        self,
        model,
        event_obj,
        exercise_boundary,
        continuation_value,
        t_index,
    ):
        return continuation_value


    def generate_lb(self, model, event_obj):
        value_array = numpy.zeros_like(model.time_grid)
        return boundaries.LowerDirichlet(value_array)


    def generate_ub(self, model, event_obj):
        return boundaries.UpperNeumann(order=1, value=1)
```

/securities/options.py

```
import numpy
from objects import boundaries
from securities import security_base
import statsmodels.api as sm


class AmericanCall(security_base.SecurityBase):
    def __init__(self, strike, maturity):
        self.strike = strike
        self.maturity = maturity
```

```
    def discount(self, model, t_index):
        return model.r[t_index]


    def coupon(self, event_obj, t_index):
        return 0.0


    def payoff(self, model, event_obj, t_max):
        return numpy.maximum(model.stock_grid[t_max] - self.strike, 0)


    def recovery(self, model, event_obj, t_index):
        return 0.0


    def estimate_exercise_boundary(
        self,
        model,
        event_obj,
        continuation_value,
        t_index,
    ):
        """
        Estimate the continuation value for LSMC.
        """
        # Initialize the array
        estimate = numpy.array(continuation_value)

        # Determine which samples are included in the regression
        if model.regression_method == "all":
            flag = numpy.ones_like(continuation_value, dtype=bool)

        elif model.regression_method == "itm":
            flag = model.stock_grid[t_index] > self.strike

        else:
            raise ValueError(f"Invalid method: {model.regression_method}")

        # Override the continuation value of these samples
        # This is faster than statsmodel API
        if flag.any():
            basis = numpy.stack([
                numpy.ones_like(model.stock_grid[t_index][flag]),
                model.stock_grid[t_index][flag],
                model.stock_grid[t_index][flag] ** 2,
                model.stock_grid[t_index][flag] ** 3,
            ], axis=1)

            result = sm.OLS(continuation_value[flag], basis).fit()
            estimate[flag] = result.fittedvalues

        return estimate


    def handle_events(
        self,
        model,
        event_obj,
        exercise_boundary,
        continuation_value,
        t_index,
    ):
        conversion_value = numpy.maximum(
            model.stock_grid[t_index] - self.strike,
            0,
        )
        continuation_value = numpy.where(
            conversion_value > exercise_boundary,
            conversion_value,
            continuation_value,
        )

        return continuation_value


    def generate_lb(self, model, event_obj):
        value_array = numpy.zeros_like(model.time_grid)
        return boundaries.LowerDirichlet(value_array)


    def generate_ub(self, model, event_obj):
        return boundaries.UpperNeumann(order=2, value=0)


class EuropeanCall(AmericanCall):
```

```python
    def __init__(self, strike, maturity):
        super().__init__(strike, maturity)


    def estimate_exercise_boundary(
        self,
        model,
        event_obj,
        continuation_value,
        t_index,
    ):
        return continuation_value


    def handle_events(
        self,
        model,
        event_obj,
        exercise_boundary,
        continuation_value,
        t_index,
    ):
        return continuation_value



class AmericanPut(security_base.SecurityBase):
    def __init__(self, strike, maturity):
        self.strike = strike
        self.maturity = maturity


    def discount(self, model, t_index):
        return model.r[t_index]


    def coupon(self, event_obj, t_index):
        return 0.0


    def payoff(self, model, event_obj, t_max):
        return numpy.maximum(self.strike - model.stock_grid[t_max], 0)


    def recovery(self, model, event_obj, t_index):
        return self.strike


    def estimate_exercise_boundary(
        self,
        model,
        event_obj,
        continuation_value,
        t_index,
    ):
        """
        Estimate the continuation value for LSMC.
        """
        # Initialize the array
        estimate = numpy.array(continuation_value)

        # Determine which samples are included in the regression
        if model.regression_method == "all":
            flag = numpy.ones_like(continuation_value, dtype=bool)

        elif model.regression_method == "itm":
            flag = model.stock_grid[t_index] < self.strike

        else:
            raise ValueError(f"Invalid method: {model.regression_method}")

        # Override the continuation value of these samples
        # This is faster than statsmodel API
        if flag.any():
            basis = numpy.stack([
                numpy.ones_like(model.stock_grid[t_index][flag]),
                model.stock_grid[t_index][flag],
                model.stock_grid[t_index][flag] ** 2,
                model.stock_grid[t_index][flag] ** 3,
            ], axis=1)

            result = sm.OLS(continuation_value[flag], basis).fit()
            estimate[flag] = result.fittedvalues

        return estimate
```

```
    def handle_events(
        self,
        model,
        event_obj,
        exercise_boundary,
        continuation_value,
        t_index,
    ):
        conversion_value = numpy.maximum(
            self.strike - model.stock_grid[t_index],
            0,
        )
        continuation_value = numpy.where(
            conversion_value > exercise_boundary,
            conversion_value,
            continuation_value,
        )

        return continuation_value


    def generate_lb(self, model, event_obj):
        return boundaries.LowerNeumann(order=2, value=0)


    def generate_ub(self, model, event_obj):
        return boundaries.UpperNeumann(order=1, value=0)


class EuropeanPut(AmericanPut):
    def __init__(self, strike, maturity):
        super().__init__(strike, maturity)


    def recovery(self, model, event_obj, t_index):
        ttm = self.maturity - model.time_grid[t_index]
        return numpy.exp(-model.r[t_index] * ttm) * self.strike


    def estimate_exercise_boundary(
        self,
        model,
        event_obj,
        continuation_value,
        t_index,
    ):
        return continuation_value


    def handle_events(
        self,
        model,
        event_obj,
        exercise_boundary,
        continuation_value,
        t_index,
    ):
        return continuation_value
```

/securities/security_base.py

```
import numpy


class SecurityBase(object):
    """
    Base class for all securities.
    """
    def discount(self, model, t_index):
        raise NotImplementedError("Child must implement")


    def coupon(self, event_obj, t_index):
        raise NotImplementedError("Child must implement")


    def payoff(self, model, event_obj, t_max):
        raise NotImplementedError("Child must implement")


    def recovery(self, model, event_obj, t_index):
        raise NotImplementedError("Child must implement")


    def generate_lb(self, model, event_obj):
        raise NotImplementedError("Child must implement")
```

```python
    def generate_ub(self, model, event_obj):
        raise NotImplementedError("Child must implement")


    def estimate_exercise_boundary(
        self,
        model,
        event_obj,
        continuation_value,
        t_index,
    ):
        raise NotImplementedError("Child must implement")


    def handle_events(
        self,
        model,
        event_obj,
        exercise_boundary,
        continuation_value,
        t_index,
    ):
        raise NotImplementedError("Child must implement")
```

/tests/run_tests.py

```python
##########################################################################################

# Navigate to inside pricer folder and run the below
# python -m tests.run_tests

##########################################################################################

import time
from tests import test_models
from tests import test_options
from tests import test_bonds
from tests import test_converts
from tests import test_analytics


def run():
    start_time = time.time()

    print("\n[test_models] running tests..")
    test_models.test()

    print("\n[test_options] running tests..")
    test_options.test()

    print("\n[test_bonds] running tests..")
    test_bonds.test()

    print("\n[test_converts] running tests..")
    test_converts.test()

    print("\n[test_analytics] running tests..")
    test_analytics.test()

    print("\nall tests passed!\n")

    end_time = time.time()
    elapsed_time = end_time - start_time
    minutes, seconds = int(elapsed_time // 60), int(elapsed_time % 60)
    print(f"Execution time: {minutes} minute(s) and {seconds} second(s)")


if __name__ == "__main__":
    run()
```

/tests/test_analytics.py

```python
import numpy
from securities import analytics
from utils import test_utils


def test_prob_simple_crr():
    """
    Handbook of Convertible Bonds p133 Table 6.12 (5 steps).
    """
    # Model parameters
    time_size = 6
    equity_vol = 0.2
```

```python
    dividend_yield = 0.02
    analytics_to_price = [
        analytics.DefaultProbability,
        analytics.OptionalConvProbability,
        analytics.MaturityConvProbability,
        analytics.CallRedemptionProbability,
        analytics.ForcedConvProbability,
        analytics.PutProbability,
        analytics.MaturityRedemptionProbability,
    ]

    # Price
    _, result_dict = test_utils.price_sample_converts_crr(
        time_size,
        equity_vol,
        dividend_yield,
        analytics_to_price,
    )

    # Make sure probabilities sum to one at each node
    for i in range(time_size):
        sum_prob = (
            result_dict["default_prob"][i]
            + result_dict["optional_conv_prob"][i]
            + result_dict["maturity_conv_prob"][i]
            + result_dict["call_redemption_prob"][i]
            + result_dict["forced_conv_prob"][i]
            + result_dict["put_prob"][i]
            + result_dict["maturity_redemption_prob"][i]
        )
        assert numpy.allclose(sum_prob, 1.0, atol=1e-4)

    # Validate probabilities on current node
    assert numpy.allclose(
        result_dict["default_prob"][0], 0.1112, atol=1e-4
    )
    assert numpy.allclose(
        result_dict["optional_conv_prob"][0], 0.0, atol=1e-4
    )
    assert numpy.allclose(
        result_dict["maturity_conv_prob"][0], 0.0, atol=1e-4
    )
    assert numpy.allclose(
        result_dict["call_redemption_prob"][0], 0.1243, atol=1e-4
    )
    assert numpy.allclose(
        result_dict["forced_conv_prob"][0], 0.1894, atol=1e-4
    )
    assert numpy.allclose(
        result_dict["put_prob"][0], 0.5750, atol=1e-4
    )
    assert numpy.allclose(
        result_dict["maturity_redemption_prob"][0], 0.0, atol=1e-4
    )


def test_prob_dense_crr():
    """
    Handbook of Convertible Bonds p133 Table 6.13 (30% vol).
    """
    # Model parameters
    time_size = 51
    equity_vol = 0.3
    dividend_yield = 0.02
    analytics_to_price = [
        analytics.DefaultProbability,
        analytics.OptionalConvProbability,
        analytics.MaturityConvProbability,
        analytics.CallRedemptionProbability,
        analytics.ForcedConvProbability,
        analytics.PutProbability,
        analytics.MaturityRedemptionProbability,
    ]

    # Price
    _, result_dict = test_utils.price_sample_converts_crr(
        time_size,
        equity_vol,
        dividend_yield,
        analytics_to_price,
    )

    # Make sure probabilities sum to one at each node
    for i in range(time_size):
        sum_prob = (
            result_dict["default_prob"][i]
            + result_dict["optional_conv_prob"][i]
```

```python
            + result_dict["maturity_conv_prob"][i]
            + result_dict["call_redemption_prob"][i]
            + result_dict["forced_conv_prob"][i]
            + result_dict["put_prob"][i]
            + result_dict["maturity_redemption_prob"][i]
        )
        assert numpy.allclose(sum_prob, 1.0, atol=1e-4)

    # Validate probabilities on current node
    assert numpy.allclose(
        result_dict["default_prob"][0], 0.1489, atol=1e-4
    )
    assert numpy.allclose(
        result_dict["optional_conv_prob"][0], 0.0053, atol=1e-4
    )
    assert numpy.allclose(
        result_dict["maturity_conv_prob"][0], 0.0091, atol=1e-4
    )
    assert numpy.allclose(
        result_dict["call_redemption_prob"][0], 0.1980, atol=1e-4
    )
    assert numpy.allclose(
        result_dict["forced_conv_prob"][0], 0.2718, atol=1e-4
    )
    assert numpy.allclose(
        result_dict["put_prob"][0], 0.2075, atol=1e-4
    )
    assert numpy.allclose(
        result_dict["maturity_redemption_prob"][0], 0.1595, atol=1e-4
    )


def test_early_exercise_crr():
    """
    Handbook of Convertible Bonds p135 Table 6.14.
    """
    # Model parameters
    time_size = 51
    equity_vol = 0.2
    analytics_to_price = [analytics.OptionalConvProbability]

    # Price
    result_dict = {}
    for dividend_yield in numpy.linspace(0, 0.05, 6):
        _, result = test_utils.price_sample_converts_crr(
            time_size,
            equity_vol,
            dividend_yield,
            analytics_to_price,
        )

        result_dict[dividend_yield] = result["optional_conv_prob"][0]

    assert numpy.allclose(
        result_dict[0.00], 0.0, atol=1e-4
    )
    assert numpy.allclose(
        result_dict[0.01], 0.0, atol=1e-4
    )
    assert numpy.allclose(
        result_dict[0.02], 0.0, atol=1e-4
    )
    assert numpy.allclose(
        result_dict[0.03], 0.004, atol=1e-4
    )
    assert numpy.allclose(
        result_dict[0.04], 0.0422, atol=1e-4
    )
    assert numpy.allclose(
        result_dict[0.05], 0.1115, atol=1e-4
    )


def test_prob_lsmc():

    default_intensity = 0.05
    stock_grid_override = [
        numpy.array([100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0]),
        numpy.array([108.17, 206.56, 175.71, 77.11, 185.29, 116.77, 99.74]),
        numpy.array([126.03, 200.26, 173.13, 104.91, 248.05, 156.57, 115.22]),
        numpy.array([99.97, 233.51, 242.36, 116.86, 308.22, 182.92, 109.52]),
    ]
    analytics_to_price = [
        analytics.DefaultProbability,
        analytics.OptionalConvProbability,
        analytics.MaturityConvProbability,
        analytics.CallRedemptionProbability,
```

```
        analytics.ForcedConvProbability,
        analytics.PutProbability,
        analytics.MaturityRedemptionProbability,
    ]
    random_seed = 42

    stock_grid, result_dict = test_utils.price_sample_converts_lsmc(
        default_intensity=default_intensity,
        stock_grid_override=stock_grid_override,
        analytics_to_price=analytics_to_price,
        random_seed=random_seed,
    )

    # Make sure probabilities sum to one at each node
    for i in range(4):
        sum_prob = (
            result_dict["default_prob"][i]
            + result_dict["optional_conv_prob"][i]
            + result_dict["maturity_conv_prob"][i]
            + result_dict["call_redemption_prob"][i]
            + result_dict["forced_conv_prob"][i]
            + result_dict["put_prob"][i]
            + result_dict["maturity_redemption_prob"][i]
        )
        assert numpy.allclose(sum_prob, 1.0, atol=1e-4)

    assert round(result_dict["default_prob"][0].mean(), 4) == 0.1204
    assert round(result_dict["optional_conv_prob"][0].mean(), 4) == 0.3878
    assert round(result_dict["maturity_conv_prob"][0].mean(), 4) == 0.4918
    assert round(result_dict["call_redemption_prob"][0].mean(), 4) == 0.0
    assert round(result_dict["forced_conv_prob"][0].mean(), 4) == 0.0
    assert round(result_dict["put_prob"][0].mean(), 4) == 0.0
    assert round(result_dict["maturity_redemption_prob"][0].mean(), 4) == 0.0


def test():
    test_prob_simple_crr()
    test_prob_dense_crr()
    test_early_exercise_crr()
    test_prob_lsmc()



/tests/test_bonds.py
import itertools
from utils import test_utils
from utils import parallel_utils


def test_bonds_crr(n_cores=8):
         maturity = [1.0, 2.0, 4.0]
         face_value = [100.0, 345.0]
         coupon_schedule = [{}, {0.5: 10.5, 0.8: 12.5}, {1.0: 12.0}]
         call_schedule = [{}]
         put_schedule = [{}]
         riskless_rate = [0.02, 0.1, 0.3]
         default_intensity = [0.0, 0.15, 0.23]
         recovery_rate = [0.0, 0.3, 0.8]

         args = itertools.product(
             maturity,
             face_value,
             coupon_schedule,
             call_schedule,
             put_schedule,
             riskless_rate,
             default_intensity,
             recovery_rate,
         )
         parallel_utils.parallelize(
                  n_cores=n_cores,
                  func=test_utils.test_bonds_crr,
                  args=list(args),
         )


def tests_bonds_pde(n_cores=8):
         maturity = [1.0, 2.0]
         face_value = [100.0, 155.0]
         coupon_schedule = [{}, {0.5: 10.5, 0.8: 12.5}, {1.0: 12.0}]
         call_schedule = [{}]
         put_schedule = [{}]
         riskless_rate = [0.02, 0.1]
         default_intensity = [0.0, 0.15, 0.23]
         recovery_rate = [0.0, 0.3]

         args = itertools.product(
```

```
            maturity,
            face_value,
            coupon_schedule,
            call_schedule,
            put_schedule,
            riskless_rate,
            default_intensity,
            recovery_rate,
        )
        parallel_utils.parallelize(
                n_cores=n_cores,
                func=test_utils.test_bonds_pde,
                args=list(args),
        )


def test():
        test_bonds_crr()
        tests_bonds_pde()
```

/tests/test_converts.py

```
import numpy
from utils import test_utils
from securities import analytics


def test_converts_crr():
        """
        Handbook of Convertible Bonds p125 example.
        """
        # Model parameters
        time_size = 6
        equity_vol = 0.2
        dividend_yield = 0.02
        analytics_to_price = [analytics.Price]

        # Price
        model_stock, model_price = test_utils.price_sample_converts_crr(
                time_size,
                equity_vol,
                dividend_yield,
                analytics_to_price,
        )

        # Expected output
        true_stock = [
                [100.0],
                [81.8731, 122.1403],
                [67.0320, 100.0, 149.1825],
                [54.8812, 81.8731, 122.1403, 182.2119],
                [44.9329, 67.0320, 100.0, 149.1825, 222.5541],
                [36.7879, 54.8812, 81.8731, 122.1403, 182.2119, 271.8282],
        ]
        true_price = [
                [103.2273],
                [101.1887, 115.9459],
                [102.5, 102.5, 128.9792],
                [101.9847, 101.9847, 105.0, 150.7695],
                [103.4181, 103.4181, 103.4181, 124.3460, 183.0433],
                [105.0, 105.0, 105.0, 105.0, 150.7695, 222.4625],
        ]

        # Validate
        for i in range(6):
                assert numpy.allclose(model_stock[i], true_stock[i], atol=0.0001)

        for i in range(6):
                assert numpy.allclose(model_price[i], true_price[i], atol=0.0001)



def test_converts_lsmc():
        """
        Handbook of Convertible Bonds p347 example.
        """
        default_intensity = 0.0
        stock_grid_override = [
                numpy.array([100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0]),
                numpy.array([102.89, 196.49, 167.15, 73.36, 176.26, 111.08, 94.88]),
                numpy.array([114.05, 181.21, 156.66, 94.93, 224.46, 141.68, 104.26]),
                numpy.array([86.04, 200.98, 208.6, 100.58, 265.28, 157.44, 94.26]),
        ]
        analytics_to_price = [analytics.Price]
        debug_mode = True
        random_seed = 42
```

```
    stock_grid, result_dict = test_utils.price_sample_converts_lsmc(
        default_intensity=default_intensity,
        stock_grid_override=stock_grid_override,
        analytics_to_price=analytics_to_price,
        debug_mode=debug_mode,
        random_seed=random_seed,
    )
    price = result_dict["solution"][0].mean()

    # Validation
    true_stock = [
        numpy.array([100., 100., 100., 100., 100., 100., 100.]),
        numpy.array([102.116, 115.964, 98.567, 81.009, 97.356, 80.908,114.533]),
        numpy.array([79.688, 75.42, 88.898, 92.799, 117.193, 83.685, 108.948]),
        numpy.array([88.962, 55.848, 85.126, 104.167, 123.634, 66.373, 100.877]),
    ]
    for i in range(4):
        assert numpy.allclose(stock_grid[i], true_stock[i], atol=0.01)

    continuation = [
        numpy.array([107.408, 190.683, 190.646, 91.923, 242.448, 143.889, 98.188]),
        numpy.array([110.679, 189.276, 196.452, 94.723, 249.831, 148.271, 101.179]),
        numpy.array([97.045, 195.04, 202.435, 97.607, 257.44, 152.787, 97.045]),
        numpy.array([100., 200.98, 208.6, 100.58, 265.28, 157.44, 100.]),
    ]

    boundary = [
        numpy.array([107.408, 190.683, 190.646, 91.923, 242.448, 143.889, 98.188]),
        numpy.array([117.698, 194.622, 222.794, 94.723, 221.047, 138.348, 101.179]),
        numpy.array([110.659, 207.078, 180.317, 97.607, 255.875, 160.47, 87.392]),
        numpy.array([100. , 200.98, 208.6, 100.58, 265.28, 157.44, 100.]),
    ]

    solution = [
        numpy.array([107.408, 190.683, 190.646, 91.923, 242.448, 143.889, 98.188]),
        numpy.array([110.679, 196.49, 196.452, 94.723, 249.831, 148.271, 101.179]),
        numpy.array([114.05, 195.04, 202.435, 97.607, 257.44, 152.787, 104.26]),
        numpy.array([100., 200.98, 208.6, 100.58, 265.28, 157.44, 100.])
    ]

    for i in range(4):
        assert numpy.allclose(result_dict["continuation"][i], continuation[i], atol=0.01)
        assert numpy.allclose(result_dict["boundary"][i], boundary[i], atol=0.01)
        assert numpy.allclose(result_dict["solution"][i], solution[i], atol=0.01)

    assert round(price, 4) == 152.1694


def test_defaultable_converts_lsmc():
    """
    Handbook of Convertible Bonds p353 example.
    """
    default_intensity = 0.05
    stock_grid_override = [
        numpy.array([100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0]),
        numpy.array([108.17, 206.56, 175.71, 77.11, 185.29, 116.77, 99.74]),
        numpy.array([126.03, 200.26, 173.13, 104.91, 248.05, 156.57, 115.22]),
        numpy.array([99.97, 233.51, 242.36, 116.86, 308.22, 182.92, 109.52]),
    ]
    analytics_to_price = [analytics.Price]
    debug_mode = True
    random_seed = 42

    stock_grid, result_dict = test_utils.price_sample_converts_lsmc(
        default_intensity=default_intensity,
        stock_grid_override=stock_grid_override,
        analytics_to_price=analytics_to_price,
        debug_mode=debug_mode,
        random_seed=random_seed,
    )
    price = result_dict["solution"][0].mean()

    # Validation
    solution = [
        numpy.array([110.126, 187.626, 194.588, 92.129, 246.395, 147.83, 100.915]),
        numpy.array([117.76, 201.715, 209.256, 98.264, 265.378, 158.605, 107.781]),
        numpy.array([126.03, 216.977, 225.146, 104.91, 285.943, 170.276, 115.22]),
        numpy.array([100., 233.51, 242.36, 116.86, 308.22, 182.92, 109.52])
    ]

    for i in range(4):
        assert numpy.allclose(result_dict["solution"][i], solution[i], atol=0.01)

    assert round(price, 4) == 154.2299
```

```
def test():
        test_converts_crr()
        test_converts_lsmc()
        test_defaultable_converts_lsmc()
```

```
/tests/test_models.py
import numpy
from utils import test_utils


def test_no_term_structure_pde():
        """
        Input riskless rate is a number.
        """
        # Parameters
        riskless_rate = 0.04
        log_discount_func = "cubic"
        atol = 1e-8
        discount_func = lambda t: numpy.exp(-riskless_rate * t)
        forward_func=lambda t: riskless_rate

        # Call the helper
        test_utils.test_term_structure_pde(
            riskless_rate=riskless_rate,
            log_discount_func=log_discount_func,
            atol=atol,
            discount_func=discount_func,
            forward_func=forward_func,
        )


def test_flat_term_structure_pde():
        """
        Log discount rate is linear which means both yield curve and
        forward curve are flat (similar to above).
        """
        # Parameters
        riskless_rate = {0.1: 0.04, 3.0: 0.04}
        log_discount_func = "linear"
        atol = 1e-8
        discount_func = lambda t: numpy.exp(-riskless_rate[0.1] * t)
        forward_func=lambda t: riskless_rate[0.1]

        # Call the helper
        test_utils.test_term_structure_pde(
            riskless_rate=riskless_rate,
            log_discount_func=log_discount_func,
            atol=atol,
            discount_func=discount_func,
            forward_func=forward_func,
        )


def test_cubic_term_structure_pde():
        """
        Log discount curve is cubic (should be in production).
        """
        # Parameters
        riskless_rate = {0.1: 0.02, 0.3: 0.03, 0.4: 0.035, 1.0: 0.05}
        log_discount_func = "cubic"
        atol = 1e-5

        # Call the helper
        test_utils.test_term_structure_pde(
            riskless_rate=riskless_rate,
            log_discount_func=log_discount_func,
            atol=atol,
        )


def test_custom_term_structure_pde():
        """
        Log discount rate is cubic but with closed-form formula.
        """
        # Parameters
        riskless_rate = {}
        log_discount_func = lambda t: -0.015 * t - 0.006 * t ** 2 + 0.0025 * t ** 3
        atol = 1e-5
        discount_func = lambda t: numpy.exp(log_discount_func(t))
        forward_func = lambda t: 0.015 + 0.012 * t - 0.0075 * t ** 2

        # Call the helper
        test_utils.test_term_structure_pde(
            riskless_rate=riskless_rate,
            log_discount_func=log_discount_func,
```

```
            atol=atol,
            discount_func=discount_func,
            forward_func=forward_func,
        )


def test():
        test_no_term_structure_pde()
        test_flat_term_structure_pde()
        test_cubic_term_structure_pde()
        test_custom_term_structure_pde()
```

/tests/test_options.py

```
import itertools
from utils import test_utils
from utils import parallel_utils


def test_european_options_crr(n_cores=8):
        stock_ladder = [100.0]
        strike_ladder = [80.0, 100.0, 120.0]
        maturity_ladder = [1.0, 2.0]
        equity_vol_ladder = [0.25, 0.75]
        riskless_rate_ladder = [0.01, 0.05, 1.0]
        dividend_yield = [0.0, 0.03]
        default_intensity = [0.0, 0.1]

        args = itertools.product(
                stock_ladder,
            strike_ladder,
            maturity_ladder,
            equity_vol_ladder,
            riskless_rate_ladder,
            dividend_yield,
            default_intensity,
        )
        parallel_utils.parallelize(
                n_cores=n_cores,
                func=test_utils.test_european_options_crr,
                args=list(args),
        )


def test_european_options_pde(n_cores=8):
        stock_ladder = [100.0]
        strike_ladder = [80.0, 100.0, 120.0]
        maturity_ladder = [1.0, 2.0]
        equity_vol_ladder = [0.25, 0.75]
        riskless_rate_ladder = [0.01, 0.05, 1.0]
        dividend_yield = [0.0, 0.03]
        default_intensity = [0.0, 0.1]

        args = itertools.product(
                stock_ladder,
            strike_ladder,
            maturity_ladder,
            equity_vol_ladder,
            riskless_rate_ladder,
            dividend_yield,
            default_intensity,
        )
        parallel_utils.parallelize(
                n_cores=n_cores,
                func=test_utils.test_european_options_pde,
                args=list(args),
        )


def test():
        test_european_options_crr()
        test_european_options_pde()
```

/utils/finite_difference.py

```
import numpy
import numba
from scipy import sparse
from scipy.sparse.linalg import splu
from scipy.sparse.linalg import spsolve


@numba.njit
def thomas(a, b, c, y):
    """
    Thomas solver for Dx = y.
```

```
    """
    n = len(y)
    c_prime = numpy.zeros(n-1)
    y_prime = numpy.zeros(n)
    x = numpy.zeros(n)

    # Forward sweep
    c_prime[0] = c[0] / b[0]
    y_prime[0] = y[0] / b[0]

    for i in range(1, n-1):
        denom = b[i] - a[i-1] * c_prime[i-1]
        c_prime[i] = c[i] / denom
        y_prime[i] = (y[i] - a[i-1] * y_prime[i-1]) / denom

    y_prime[n-1] = (y[n-1] - a[n-2] * y_prime[n-2]) / (b[n-1] - a[n-2] * c_prime[n-2])

    # Backward substitution
    x[n-1] = y_prime[n-1]
    for i in range(n-2, -1, -1):
        x[i] = y_prime[i] - c_prime[i] * x[i+1]

    return x


class ThetaScheme:
    def __init__(self, theta, solver):
        self.theta = theta
        self.solver = self.get_solver(solver)


    def get_solver(self, solver):
        """
        Avoid checking if else at every iteration.
        """
        if solver == "spsolve":
            return lambda D, y: spsolve(D, y)
        elif solver == "splu":
            return lambda D, y: splu(D).solve(y)
        else:
            return lambda D, y: thomas(
                D.diagonal(-1), D.diagonal(0), D.diagonal(1), y
            )


    def apply_boundary_conditions(
        self, lower_boundary, upper_boundary, model, t_index, a, b, c, B
    ):
        """
        Modify edges of tridiagonal matrix (same for LHS and RHS).
        """
        b[0], c[0], B[0] = lower_boundary.update_matrix(model, t_index, a, b, c)
        a[-1], b[-1], B[-1] = upper_boundary.update_matrix(model, t_index, a, b, c)

        return a, b, c, B


    def backward_lhs(
        self,
        theta,
        model,
        price_obj,
        lower_boundary,
        upper_boundary,
        t_index,
    ):
        """
        LHS of backward Komogorov equations.
        """
        a = theta * (model.r[t_index] - model.dividend_yield + model.jtd_grid[t_index]
            - 0.5 * model.sigma[t_index] ** 2) * model.dt / (2 * model.dX
            ) - 0.5 * theta * model.sigma[t_index] ** 2 * model.dt / model.dX ** 2

        b = 1 + theta * (price_obj.discount(model, t_index) + model.jtd_grid[t_index]
            ) * model.dt + theta * model.sigma[t_index] ** 2 * model.dt / model.dX ** 2

        c = -theta * (model.r[t_index] - model.dividend_yield + model.jtd_grid[t_index]
            - 0.5 * model.sigma[t_index] ** 2) * model.dt / (2 * model.dX
            ) - 0.5 * theta * model.sigma[t_index] ** 2 * model.dt / model.dX ** 2

        B = numpy.zeros(model.space_size)

        a, b, c, B = self.apply_boundary_conditions(
            lower_boundary, upper_boundary, model, t_index, a, b, c, B
        )

        return a, b, c, B
```

```
def backward_rhs(
    self,
    theta,
    model,
    price_obj,
    lower_boundary,
    upper_boundary,
    t_index,
):
    """
    RHS of backward Komogorov equation (identity matrix if for implicit).
    """
    a = -(1 - theta) * (model.r[t_index] - model.dividend_yield +
        model.jtd_grid[t_index]- 0.5 * model.sigma[t_index] ** 2) * model.dt / (2 * model.dX
        ) + 0.5 * (1 - theta) * model.sigma[t_index] ** 2 * model.dt / model.dX ** 2

    b = 1 - (1 - theta) * (price_obj.discount(model, t_index) + model.jtd_grid[t_index]
        ) * model.dt - (1 - theta) * model.sigma[t_index] ** 2 * model.dt / model.dX ** 2

    c = (1 - theta) * (model.r[t_index] - model.dividend_yield +
        model.jtd_grid[t_index] - 0.5 * model.sigma[t_index] ** 2) * model.dt / (2 * model.dX
        ) + 0.5 * (1 - theta) * model.sigma[t_index] ** 2 * model.dt / model.dX ** 2

    B = numpy.zeros(model.space_size)

    a, b, c, B = self.apply_boundary_conditions(
        lower_boundary, upper_boundary, model, t_index, a, b, c, B
    )

    return a, b, c, B


def backward_komogorov(
    self,
    model,
    price_obj,
    event_obj,
    t_start,
    t_index,
    lower_boundary,
    upper_boundary,
    solution_array,
):
    """
    Construct tridiagonal matrix and step backward.
    """
    theta = self.get_theta(t_start, t_index)

    # LHS is at t
    a, b, c, B = self.backward_lhs(
        theta=theta,
        model=model,
        price_obj=price_obj,
        lower_boundary=lower_boundary,
        upper_boundary=upper_boundary,
        t_index=t_index,
    )
    D = sparse.diags(
        [a[1:], b, c[:-1]],
        [-1, 0, 1],
        shape=(model.space_size, model.space_size),
    ).tocsc()

    # RHS is at t + 1
    a_s, b_s, c_s, B_s = self.backward_rhs(
        theta=theta,
        model=model,
        price_obj=price_obj,
        lower_boundary=lower_boundary,
        upper_boundary=upper_boundary,
        t_index=t_index + 1,
    )
    D_s = sparse.diags(
        [a_s[1:], b_s, c_s[:-1]],
        [-1, 0, 1],
        shape=(model.space_size, model.space_size),
    ).tocsc()

    # Jump term
    R = model.dt * (
        theta * model.jtd_grid[t_index] * price_obj.recovery(model, event_obj, t_index)
        + (1 - theta) * model.jtd_grid[t_index + 1] * price_obj.recovery(model, event_obj, t_index + 1)
    )

    # Solve the system
```

```python
        return self.solver(D, D_s @ solution_array + B_s - B + R)


    def forward_lhs(self, theta, model, at, bt, t_index):
        """
        LHS of forward Komogorov equation.
        """
        lam = model.jtd_func(
            at, model.stock_price, model.stock_grid[t_index], model.equity_to_credit,
        )

        lam_p = model.jtd_deriv_func(
            at, model.stock_price, model.stock_grid[t_index], model.equity_to_credit,
        )

        a = -theta * (model.r[t_index] - model.dividend_yield + lam
            - 0.5 * bt ** 2) * model.dt / (2 * model.dX
            ) - 0.5 * theta * bt ** 2 * model.dt / model.dX ** 2

        b = 1 + theta * (model.r[t_index] + lam + model.stock_grid[t_index]
            * lam_p) * model.dt + theta * bt ** 2 * model.dt / model.dX ** 2

        c = theta * (model.r[t_index] - model.dividend_yield + lam
            - 0.5 * bt ** 2) * model.dt / (2 * model.dX
            ) - 0.5 * theta * bt ** 2 * model.dt / model.dX ** 2

        return a, b, c


    def forward_rhs(self, theta, model, at, bt, t_index):
        """
        RHS of forward Komogorov equation.
        """
        lam = model.jtd_func(
            at, model.stock_price, model.stock_grid[t_index], model.equity_to_credit,
        )

        lam_p = model.jtd_deriv_func(
            at, model.stock_price, model.stock_grid[t_index], model.equity_to_credit,
        )

        a = (1 - theta) * (model.r[t_index] - model.dividend_yield + lam
            - 0.5 * bt ** 2) * model.dt / (2 * model.dX) + 0.5 * (1 - theta
            ) * bt ** 2 * model.dt / model.dX ** 2

        b = 1 - (1 - theta) * (model.r[t_index] + lam +
            model.stock_grid[t_index] * lam_p) * model.dt - (1 - theta
            ) * bt ** 2 * model.dt / model.dX ** 2

        c = -(1 - theta) * (model.r[t_index] - model.dividend_yield + lam
            - 0.5 * bt ** 2) * model.dt / (2 * model.dX) + 0.5 * (1 - theta
            ) * bt ** 2 * model.dt / model.dX ** 2

        return a, b, c


    def forward_komogorov(self, model, at, bt, t_start, t_index, solution_array):
        """
        Construct tridiagonal matrix and step forward.
        """
        theta = self.get_theta(t_start, t_index)

        # LHS is at t
        a, b, c = self.forward_lhs(theta, model, at, bt, t_index)
        D = sparse.diags(
            [a[1:], b, c[:-1]],
            [-1, 0, 1],
            shape=(model.space_size, model.space_size),
        ).tocsc()

        # RHS is at t - 1
        a_s, b_s, c_s = self.forward_rhs(theta, model, at, bt, t_index - 1)
        D_s = sparse.diags(
            [a_s[1:], b_s, c_s[:-1]],
            [-1, 0, 1],
            shape=(model.space_size, model.space_size),
        ).tocsc()

        # Solve the system
        return self.solver(D, D_s @ solution_array)


class ImplicitScheme(ThetaScheme):
    def __init__(self, solver):
        super().__init__(1.0, solver)
```

```
    def get_theta(self, t_start, t_index):
        return self.theta


class CrankNicolsonScheme(ThetaScheme):
    def __init__(self, solver):
        super().__init__(0.5, solver)


    def get_theta(self, t_start, t_index):
        return self.theta


class RannacherScheme(ThetaScheme):
    def __init__(self, solver):
        super().__init__(numpy.nan, solver)


    def get_theta(self, t_start, t_index):
        return 1.0 if abs(t_index - t_start) < 5 else 0.5
```

/utils/parallel_utils.py

```
# Note that the function that will be called MUST be defined inside a Python module, e.g.
# quantpy/workers.py and CANNOT be defined inside jupyter notebook, otherwise it will hang.
# https://stackoverflow.com/questions/23641475/multiprocessing-working-in-python-but-not-in-ipython/23641560#23641560

from multiprocessing import Pool


MAX_CORES = 10


def parallelize(n_cores, func, args):
    """
    Apply a function on multiple sets of parameters.

    >>> print(MAX_CORES)
    7
    """
    assert n_cores <= MAX_CORES, f"Exceed limit: {n_cores} > {MAX_CORES}"
    assert n_cores <= len(args), f"Some cores will be idle: {n_cores} > {len(args)}"

    with Pool(processes=n_cores) as pool:
        out = pool.starmap(func, args)

    return out


if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

/utils/parsing_utils.py

```
import numpy
import pandas
from scipy.interpolate import UnivariateSpline


def parse_curve(time_grid, curve, func="step"):
    """
    Helper to parse the curves.
    """
    if isinstance(curve, (int, float)):
        return numpy.full_like(time_grid, curve)

    # Piecewise constant
    if func == "step":
        array = numpy.full_like(time_grid, numpy.nan)
        for t, s in curve.items():
            t_index = numpy.searchsorted(time_grid, t)
            array[t_index] = s

        return pandas.Series(array).bfill().ffill().to_numpy()

    # Piecewise linear
    elif func == "linear":
        tenors = sorted(curve.keys())
        values = [curve[t] for t in tenors]

        return numpy.interp(time_grid, tenors, values)

    # Cubic spline
    elif func == "cubic":
        tenors = sorted(curve.keys())
```

```python
        values = [curve[t] for t in tenors]

        return UnivariateSpline(tenors, values, s=0)(time_grid)

    # Custom function
    else:
        return func(time_grid)


def parse_schedule(time_grid, schedule):
    """
    Helper to parse the schedules.
    """
    # Initialize arrays
    flag_array = numpy.zeros_like(time_grid, dtype=bool)
    trigger_array = numpy.full_like(time_grid, numpy.nan, dtype=float)
    price_array = numpy.full_like(time_grid, numpy.nan, dtype=float)

    # Populate information
    for time, details in schedule.items():

        # Key is a timestamp such as coupon schedule
        if isinstance(time, (int, float)):
            start_idx = numpy.searchsorted(time_grid, time)
            end_idx = start_idx

        # Key is a period such as call, put and conversion schedules
        elif isinstance(time, (tuple, list)):
            start_idx = numpy.searchsorted(time_grid, time[0])
            end_idx = numpy.searchsorted(time_grid, time[1])

        else:
            raise ValueError(f"Invalid type: {time}")

        # Value is the amount such as coupon, put and conversion schedules
        if isinstance(details, (int, float)):
            trigger, price = numpy.nan, details

        # Value is trigger, amount tuple such as call schedule and dividend protection
        elif isinstance(details, (tuple, list)):
            trigger, price = details[0], details[1]

        else:
            raise ValueError(f"Invalid type: {details}")

        flag_array[start_idx:end_idx + 1] = True
        trigger_array[start_idx:end_idx + 1] = trigger
        price_array[start_idx:end_idx + 1] = price

    return flag_array, trigger_array, price_array


def accrued_interest(time_grid, coupon_schedule):
    """
    Helper to calculate accrued interest.
    """
    # Parse first
    has_coupon, _, coupon = parse_schedule(
        time_grid,
        coupon_schedule,
    )

    # The timestamp of the coupon dates
    coupon_time = numpy.where(has_coupon, time_grid, numpy.nan)

    # Previous coupon
    prev_time = pandas.Series(coupon_time).ffill().fillna(time_grid[0]).to_numpy()

    # Next coupon
    next_time = pandas.Series(coupon_time).bfill().fillna(time_grid[-1]).to_numpy()
    next_coupon = pandas.Series(coupon).bfill().fillna(0).to_numpy()

    # Calculation
    accrued = numpy.divide(
        next_coupon * (time_grid - prev_time),
        (next_time - prev_time),
        where=~has_coupon,
        out=numpy.zeros_like(time_grid),
    )

    return accrued


def pv_future_cashflow(model, security, coupon):
    """
    Each node represents the PV of all future cashflows at that node.
    """
```

```python
    # Initialize array
    bond_floor = numpy.zeros_like(model.time_grid)

    # Start from maturity date and move backwards
    t_max = numpy.searchsorted(model.time_grid, security.maturity)
    bond_floor[t_max] = security.face_value + coupon[t_max]

    for t_index in range(t_max - 1, -1, -1):
        # Discount from next to current step
        bond_floor[t_index] = (
            numpy.exp(-model.r[t_index] * model.dt)
            * bond_floor[t_index + 1]
            + coupon[t_index]
        )

    return bond_floor
```

/utils/pricing_utils.py

```python
import numpy
from scipy.stats import norm
from scipy.optimize import brentq


def bs_d1_d2(S, K, T, r, q, sigma):
    """
    Black-Schoels N(d1) and N(d2).
    """
    d1 = (numpy.log(S / K) + (r - q + 0.5 * sigma ** 2) * T) / (sigma * numpy.sqrt(T))
    d2 = d1 - sigma * numpy.sqrt(T)
    return d1, d2


def bs_call(S, K, T, r, q, l, sigma):
    """
    Black-Scholes call option price.

    If l != 0 then use Merton 1976 constant jump-to-zero.
    """
    d1, d2 = bs_d1_d2(S, K, T, r + l, q, sigma)
    return numpy.exp(-q * T) * S * norm.cdf(d1) - K * numpy.exp(-(r + l) * T) * norm.cdf(d2)


def bs_put(S, K, T, r, q, l, sigma):
    """
    Black-Scholes put option price.

    If l != 0 then use Merton 1976 constant jump-to-zero.
    """
    d1, d2 = bs_d1_d2(S, K, T, r + l, q, sigma)
    return K * numpy.exp(-(r + l) * T) * norm.cdf(-d2) - numpy.exp(
        -q * T) * S * norm.cdf(-d1) + K * (1 - numpy.exp(-l * T)) * numpy.exp(-r * T)


def bs_put_pcp(S, K, T, r, q, l, sigma):
    """
    Back out Black-Scholes put option price from Put-Call-Parity.
    """
    c = bs_call(S, K, T, r, q, l, sigma)
    return c - S * numpy.exp(-q * T) + K * numpy.exp(-r * T)


def bs_cvol(price, S, K, T, r, q):
    """
    Iterative solver for Black-Scholes implied volatility.
    """
    def obj_fun(sigma):
        return price - bs_call(S, K, T, r, q, 0.0, sigma)
    return brentq(obj_fun, 1e-6, 5.0)


def bs_pvol(price, S, K, T, r, q):
    """
    Iterative solver for Black-Scholes implied volatility.
    """
    def obj_fun(sigma):
        return price - bs_put(S, K, T, r, q, 0.0, sigma)
    return brentq(obj_fun, 1e-6, 5.0)
```

/utils/security_utils.py

```python
import copy


def disable_conversion_terms(security):
    """
```

```
    Will not cast to a different class.
    """
    security = copy.deepcopy(security)
    security.conversion_schedule = {}

    return security


def disable_call_terms(security):
    """
    Will not cast to a different class.
    """
    security = copy.deepcopy(security)
    security.call_schedule = {}

    return security


def disable_put_terms(security):
    """
    Will not cast to a different class.
    """
    security = copy.deepcopy(security)
    security.put_schedule = {}

    return security
```

```
/utils/test_utils.py
import copy
import numpy
from objects import events
from models import binomial_tree, monte_carlo, pde
from securities import bonds, options, converts
from utils import pricing_utils


def test_european_options_crr(
    stock_price,
    strike,
    maturity,
    equity_vol,
    riskless_rate,
    dividend_yield,
    default_intensity,
):
    # Set up model object
    time_size = 500
    spline_size = 50
    equity_to_credit = 0.0

    params = locals()

    crr_model = binomial_tree.BinomialTree(
        max_years=maturity,
        stock_price=stock_price,
        equity_vol=equity_vol,
        riskless_rate=riskless_rate,
        dividend_yield=dividend_yield,
        spline_size=spline_size,
        time_size=time_size,
        default_intensity=default_intensity,
        equity_to_credit=equity_to_credit,
    )

    # Set up security objects
    european_call = options.EuropeanCall(strike=strike, maturity=maturity)
    european_put = options.EuropeanPut(strike=strike, maturity=maturity)

    # Call price
    crr_call = crr_model.price(european_call)
    bs_call = pricing_utils.bs_call(
        crr_model.stock_grid[0],
        strike,
        maturity,
        riskless_rate,
        dividend_yield,
        default_intensity,
        equity_vol,
    )

    # Put price
    crr_put = crr_model.price(european_put)
    bs_put = pricing_utils.bs_put(
        crr_model.stock_grid[0],
        strike,
```

```
        maturity,
        riskless_rate,
        dividend_yield,
        default_intensity,
        equity_vol,
    )
    pcp_put = pricing_utils.bs_put_pcp(
        crr_model.stock_grid[0],
        strike,
        maturity,
        riskless_rate,
        dividend_yield,
        default_intensity,
        equity_vol,
    )

    # Validation
    assert numpy.allclose(crr_call[0], bs_call, atol=0.05), params
    assert numpy.allclose(crr_put[0], bs_put, atol=0.05), params
    assert numpy.allclose(crr_put[0], pcp_put, atol=0.05), params


def test_european_options_pde(
    stock_price,
    strike,
    maturity,
    equity_vol,
    riskless_rate,
    dividend_yield,
    default_intensity,
):
    # Set up model object
    time_size = 1700
    space_size = 1700
    min_space = 80
    max_space = 80
    equity_to_credit = 0.0

    params = locals()

    pde_model = pde.PDE(
        max_years=maturity,
        stock_price=stock_price,
        equity_vol=equity_vol,
        riskless_rate=riskless_rate,
        dividend_yield=dividend_yield,
        time_size=time_size,
        space_size=space_size,
        min_space=min_space,
        max_space=max_space,
        default_intensity=default_intensity,
        equity_to_credit=equity_to_credit,
    )

    # Set up security objects
    european_call = options.EuropeanCall(strike=strike, maturity=maturity)
    european_put = options.EuropeanPut(strike=strike, maturity=maturity)

    # Call price
    pde_call = pde_model.price(european_call)[0]
    bs_call = pricing_utils.bs_call(
        pde_model.stock_grid[0],
        strike,
        maturity,
        riskless_rate,
        dividend_yield,
        default_intensity,
        equity_vol,
    )

    # Put price
    pde_put = pde_model.price(european_put)[0]
    bs_put = pricing_utils.bs_put(
        pde_model.stock_grid[0],
        strike,
        maturity,
        riskless_rate,
        dividend_yield,
        default_intensity,
        equity_vol,
    )

    # Validation
    assert numpy.allclose(pde_call, bs_call, atol=0.05), params
    assert numpy.allclose(pde_put, bs_put, atol=0.05), params
```

```python
def test_bonds_crr(
    maturity,
    face_value,
    coupon_schedule,
    call_schedule,
    put_schedule,
    riskless_rate,
    default_intensity,
    recovery_rate,
):
    # Set up model object
    time_size = 1000
    spline_size = 50
    stock_price = 100.0
    equity_vol = 0.35
    dividend_yield = 0.01
    equity_to_credit = 0.0
    recovery_type = "par"

    params = locals()

    crr_model = binomial_tree.BinomialTree(
        max_years=maturity,
        stock_price=stock_price,
        equity_vol=equity_vol,
        riskless_rate=riskless_rate,
        dividend_yield=dividend_yield,
        spline_size=spline_size,
        time_size=time_size,
        default_intensity=default_intensity,
        equity_to_credit=equity_to_credit,
    )

    # Set up security objects
    coupon_bond = bonds.Bond(
        maturity=maturity,
        face_value=face_value,
        recovery_rate=recovery_rate,
        recovery_type=recovery_type,
        coupon_schedule=coupon_schedule,
        call_schedule=call_schedule,
        put_schedule=put_schedule,
    )
    crr_bond = crr_model.price(coupon_bond)

    risky_pv = coupon_bond.get_risky_bond_floor(
        crr_model,
        events.Event(crr_model, coupon_bond),
    )

    # Validation
    assert numpy.allclose(crr_bond[0], risky_pv[0], atol=0.01), params


def test_bonds_pde(
    maturity,
    face_value,
    coupon_schedule,
    call_schedule,
    put_schedule,
    riskless_rate,
    default_intensity,
    recovery_rate,
):
    # Set up model object
    time_size = 1000
    stock_price = 100.0
    equity_vol = 0.35
    dividend_yield = 0.01
    cash_dividend = {}
    equity_to_credit = 0.0
    recovery_type = "par"
    time_size = 2000
    space_size = 2000
    min_space = 80
    max_space = 80

    params = locals()

    pde_model = pde.PDE(
        max_years=maturity,
        stock_price=stock_price,
        equity_vol=equity_vol,
        riskless_rate=riskless_rate,
        dividend_yield=dividend_yield,
        cash_dividend=cash_dividend,
        time_size=time_size,
```

```python
        space_size=space_size,
        min_space=min_space,
        max_space=max_space,
        default_intensity=default_intensity,
        equity_to_credit=equity_to_credit,
    )

    # Set up security objects
    coupon_bond = bonds.Bond(
        maturity=maturity,
        face_value=face_value,
        recovery_rate=recovery_rate,
        recovery_type=recovery_type,
        coupon_schedule=coupon_schedule,
        call_schedule=call_schedule,
        put_schedule=put_schedule,
    )

    # Get rid of lower boundary because the boundary condition is
    # designed for jump diffusion model and is pinned at recovery
    pde_bond = numpy.interp(
        numpy.linspace(stock_price / 3.0, stock_price * 3.0, 300),
        pde_model.stock_grid[0],
        pde_model.price(coupon_bond)[0],
    )

    risky_pv = coupon_bond.get_risky_bond_floor(
        pde_model,
        events.Event(pde_model, coupon_bond),
    )

    # Validation
    assert numpy.allclose(pde_bond, risky_pv[0], atol=0.01), params


def price_sample_converts_crr(
    time_size,
    equity_vol,
    dividend_yield,
    analytics_to_price,
):
    """
    Handbook of Convertible Bonds p125 example.
    """
    # Model parameters
    spline_size = 1
    max_years = 5.0
    stock_price = 100.0
    riskless_rate = 0.03
    default_intensity = 0.05
    equity_to_credit = 0.0

    # Converts paramaters
    maturity = 5.0
    face_value = 100.0
    conversion_schedule = {(0, 5.0): 0.8}
    coupon_schedule = {1: 5.0, 2: 5.0, 3: 5.0, 4: 5.0, 5: 5.0}
    call_schedule = {(3, 4): (120, 100.0)}
    call_notice_period = 0
    put_schedule = {(2, 2): 102.5}
    recovery_rate = 0.3
    recovery_type = "pv"
    call_cushion = 0.0
    call_cushion_prob_1m = 0.0
    dividend_protection= {}

    # Price
    model = binomial_tree.BinomialTree(
        spline_size=spline_size,
        time_size=time_size,
        max_years=max_years,
        stock_price=stock_price,
        equity_vol=equity_vol,
        riskless_rate=riskless_rate,
        dividend_yield=dividend_yield,
        default_intensity=default_intensity,
        equity_to_credit=equity_to_credit,
    )
    conv_bond = converts.ConvertibleBond(
        maturity=maturity,
        face_value=face_value,
        coupon_schedule=coupon_schedule,
        recovery_rate=recovery_rate,
        recovery_type=recovery_type,
        conversion_schedule=conversion_schedule,
        call_schedule=call_schedule,
        call_notice_period=call_notice_period,
```

```
        put_schedule=put_schedule,
        call_cushion=call_cushion,
        call_cushion_prob_1m=call_cushion_prob_1m,
        dividend_protection=dividend_protection,
    )

    result = model.price(
        conv_bond,
        analytics_to_price=analytics_to_price,
    )

    return model.stock_grid, result


def price_sample_converts_lsmc(
    default_intensity,
    stock_grid_override,
    analytics_to_price,
    debug_mode=False,
    random_seed=42,
):
    # Model parameters
    num_paths = 7
    max_years = 3.0
    stock_price = 100.0
    equity_vol = 0.2
    riskless_rate = 0.03
    dividend_yield = 0.05
    cash_dividend = {}
    time_size = 4
    equity_to_credit = 0.0
    regression_method = "itm"

    # Security parameters
    maturity = 3.0
    face_value = 100.0
    coupon_schedule = {}
    recovery_rate = 0.3
    recovery_type = "par"
    conversion_schedule = {(1, 3): 1.0}
    call_schedule = {}
    call_notice_period = 0
    put_schedule = {}
    call_cushion = 0.0
    call_cushion_prob_1m = 0.0
    dividend_protection = {}

    model = monte_carlo.MonteCarlo(
        num_paths=num_paths,
        max_years=max_years,
        stock_price=stock_price,
        equity_vol=equity_vol,
        riskless_rate=riskless_rate,
        dividend_yield=dividend_yield,
        cash_dividend=cash_dividend,
        time_size=time_size,
        default_intensity=default_intensity,
        equity_to_credit=equity_to_credit,
        random_seed=random_seed,
        regression_method=regression_method,
    )
    conv_bond = converts.ConvertibleBond(
        maturity=maturity,
        face_value=face_value,
        coupon_schedule=coupon_schedule,
        recovery_rate=recovery_rate,
        recovery_type=recovery_type,
        conversion_schedule=conversion_schedule,
        call_schedule=call_schedule,
        call_notice_period=call_notice_period,
        put_schedule=put_schedule,
        call_cushion=call_cushion,
        call_cushion_prob_1m=call_cushion_prob_1m,
        dividend_protection=dividend_protection,
    )

    stock_grid = copy.deepcopy(model.stock_grid)

    # Manually override the simulated paths
    # No need to override the jump intensity grid since it is unused
    model.stock_grid = stock_grid_override

    result_dict = model.price(
        conv_bond,
        analytics_to_price=analytics_to_price,
        debug_mode=debug_mode,
    )
```

```python
    return stock_grid, result_dict


def test_term_structure_pde(
    riskless_rate,
    log_discount_func,
    atol,
    discount_func=None,
    forward_func=None,
):
    # Model parameters
    max_years = 1.0
    stock_price = 50
    equity_vol = 0.3
    dividend_yield = 0.02
    time_size = 3000
    space_size = 100
    min_space = 120.0
    max_space = 120.0

    # Build model
    model = pde.PDE(
        max_years=max_years,
        stock_price=stock_price,
        equity_vol=equity_vol,
        riskless_rate=riskless_rate,
        log_discount_func=log_discount_func,
        dividend_yield=dividend_yield,
        time_size=time_size,
        space_size=space_size,
        min_space=min_space,
        max_space=max_space,
    )

    # Test discount curve against closed-form if exists
    if discount_func:
        P = discount_func(model.time_grid)
        assert numpy.allclose(P, model.P, atol=atol, rtol=0)

    # Test forward curve against closed-form if exists
    if forward_func:
        r = forward_func(model.time_grid)
        assert numpy.allclose(r, model.r, atol=atol, rtol=0)

    # Integral of forward curve from 0 to t
    integral = numpy.zeros_like(model.r)
    integral[1:] = numpy.cumsum(model.r[:-1] * model.dt)

    # Calculate discount curve from forward curve
    P = numpy.exp(-integral)

    # Calculate yield curve from forward curve
    R = numpy.zeros(model.time_size)
    R[1:] = integral[1:] / model.time_grid[1:]
    R[0] = R[1]

    # Test forward curve is consistent with discount curve
    assert numpy.allclose(P, model.P, atol=atol, rtol=0)

    # Test forward curve is consistent with yield curve
    if isinstance(riskless_rate, (int, float)):
        assert numpy.allclose(R, riskless_rate, atol=atol, rtol=0)
    else:
        for tenor, market_yield in riskless_rate.items():
            if tenor > model.max_years:
                continue
            t_index = numpy.searchsorted(model.time_grid, tenor)
            assert numpy.allclose(R[t_index], market_yield, atol=atol, rtol=0)

    # Test PDE solution of zero bond match discount curve
    for maturity in [0.2, 0.4, 0.6, 0.8]:
        t_index = numpy.searchsorted(model.time_grid, maturity)
        bond = bonds.Bond(
            maturity=maturity,
            face_value=1.0,
            coupon_schedule={},
            call_schedule={},
            put_schedule={},
            recovery_rate=0.0,
            recovery_type="par",
        )
        solution = numpy.interp(
            stock_price,
            model.stock_grid[0],
            model.price(bond)[0],
        )
```

```
assert numpy.allclose(solution, model.P[t_index], atol=atol, rtol=0)
```