A Appendices

A Code

Written code for models and testing methods.

A.1 DCC-GARCH

```
class DCC_GARCH:
    """Class that generates scenarios with dcc_garch model."""

    def __init__(self, M: int, N: int, P: int, Q: int, data: pd.DataFrame) -> None:
        self.M = M
        self.N = N
        self.P = P
        self.Q = Q
        self.T = len(data)
        self.n = len(data.columns)
        self.data = data

    def garch_var(self, params_garch: Any, data: np.array) -> np.array:
        """Calculate variance for one asset over the whole data set."""
        alpha0 = params_garch[0]
        alpha = params_garch[1 : self.P + 1]
        beta = params_garch[self.P + 1 :]
        var_t = np.zeros(self.T)
        lag = max(self.Q, self.P)
        for t in range(0, self.T):
            if t < lag:
                var_t[t] = data[t] ** 2
            else:
                if self.P == 1:
                    var_alph = alpha * (data[t - 1] ** 2)
                if self.Q == 1:
                    var_beta = beta * var_t[t - 1]
                else:
                    var_alph = np.dot(alpha, data[t - self.Q : t] ** 2)
                    var_beta = np.dot(beta, var_t[t - self.P : t])
                var_t[t] = alpha0 + var_alph + var_beta
        assert np.all(var_t > 0)
        assert not np.isnan(var_t).any()
        return var_t
```

```python

    def garch_loglike(self, params_garch: Any, data: np.array) -> Any:
        """Calculate loglikelihood for each asset separatly."""
        var_t = self.garch_var(params_garch, data)
        Loglike = np.sum(-np.log(var_t) - (data ** 2) / var_t)
        return -Loglike

    def garch_fit(self, data: np.array) -> Any:
        """Minimize the negative loglikelihood to estimate the parameters."""
        total_parameters = 1 + self.P + self.Q
        start_params = np.zeros(total_parameters)
        start_params[0] = 0.01
        start_params[1 : self.P + 1] = 0.01
        start_params[self.P + 1 :] = 0.97
        bonds = []
        for _i in range(0, total_parameters):
            bonds.append((1e-6, 0.9999))
        # If you would want a working algorithm for P,Q>1 this could be used but chosing ...
            the start params is notoriously hard
        # if max(self.P,self.Q)>1:
        # constraint = {'type': 'ineq', 'fun': lambda x: 1 - sum(x[1:self.P+1]) - ...
            sum(x[self.P+1:])}
        # res = minimize(self.garch_loglike, (start_params), args=(data), bounds= bonds, ...
            constraints= constraint, options={'disp':True})
        res = minimize(self.garch_loglike, (start_params), args=(data), bounds=bonds)
        return res.x

    def dcc_covar(self, data: pd.DataFrame, params_dcc: Any, D_t: np.array) -> Any:
        """Calculate the dynamic conitional correlation matrix and residuals."""
        # parameters a and b
        a = params_dcc[: self.M]
        b = params_dcc[self.M :]
        # calculation of residuals and Q_bar (constant conditional correlation matrix)
        et = np.zeros((self.n, self.T))
        Q_bar = np.zeros((self.n, self.n))
        for t in range(0, self.T):
            et[:, t] = np.matmul(np.linalg.inv(np.diag(D_t[t, :])), ...
                np.transpose(data.iloc[t, :]))
            et_i = et[:, t].reshape((self.n, 1))
            Q_bar = Q_bar + np.matmul(et_i, et_i.T)
        Q_bar = (1 / self.T) * Q_bar
        # calculation of Q_t, the building stone of Rt, the dynamic conditional ...
            correlation matrix
```

```
        lag = max(self.M, self.N)
        Q_tn = np.zeros((self.T, self.n, self.n))
        R = np.zeros((self.T, self.n, self.n))
        Q_tn[0] = np.matmul(np.transpose(data.iloc[0, :] / 2), data.iloc[0, :] / 2)
        for t in range(1, self.T):
            # start values, niet van toepassing voor M=N=1, source is the dcc code on ...
                which this structure is based
            if t < lag:
                Q_tn[t] = np.matmul(np.transpose(data.iloc[t, :] / 2), data.iloc[t, :] / 2)
                assert not np.isnan(Q_tn[t]).any()
            if lag == 1:
                et_i = et[:, t - 1].reshape((self.n, 1))
                Q_tn[t] = (1 - a - b) * Q_bar + a * np.matmul(et_i, et_i.T) + b * Q_tn[t - 1]
                assert not np.isnan(Q_tn[t]).any()
            else:
                a_sum = np.zeros((self.n, self.n))
                b_sum = np.zeros((self.n, self.n))
                if self.M == 1:
                    a_sum = a * np.matmul(
                        et[:, t - 1].reshape((self.n, 1)),
                        np.transpose(et[:, t - 1].reshape((self.n, 1))),
                    )
                if self.N == 1:
                    b_sum = b * Q_tn[t - 1]
                else:
                    for m in range(1, self.M):
                        a_sum = a_sum + a[m - 1] * np.matmul(
                            et[:, t - m].reshape((self.n, 1)),
                            np.transpose(et[:, t - m].reshape((self.n, 1))),
                        )
                    for n in range(1, self.N):
                        b_sum = b_sum + b[n - 1] * Q_tn[t - n]
                Q_tn[t] = (1 - np.sum(a) - np.sum(b)) * Q_bar + a_sum + b_sum
            Q_star = np.diag(np.sqrt(np.diagonal(Q_tn[t])))
            R[t] = np.matmul(np.matmul(np.linalg.inv(Q_star), Q_tn[t]), np.linalg.inv(Q_star))
        self.Q_bar = Q_bar
        self.Q_tn = Q_tn
        self.et = et
        return R, et

    def dcc_loglike(self, params_dcc: Any, data: pd.DataFrame, D_t: np.array) -> Any:
        """Calculate loglikelihood for dcc estimation."""
        Loglike = 0
```

```python
        R, et = self.dcc_covar(data, params_dcc, D_t)
        for t in range(1, self.T):
            et_i = et[:, t].reshape((self.n, 1))
            residual_part = np.matmul(et_i.T, np.matmul(np.linalg.inv(R[t]), et_i))
            determinant_part = np.log(np.linalg.det(R[t]))
            assert determinant_part != 0
            Loglike = Loglike + determinant_part + residual_part[0][0]
        return Loglike

    def dcc_fit(self, data: pd.DataFrame) -> Any:
        """Fit the parameters for the dynamic conditional correlation."""
        # Estimation of garch params and calculation of the variances
        D_t = np.zeros((self.T, self.n))
        par_garch_n = np.zeros((self.n, 1 + self.P + self.Q))
        for i in range(0, self.n):
            par_garch_i = self.garch_fit(data.iloc[:, i].to_numpy())
            par_garch_n[i, :] = par_garch_i
            D_t[:, i] = np.sqrt(self.garch_var(par_garch_i, data.iloc[:, i].to_numpy()))
        # Estimation of dcc params, both low starting values to give the algorithm more ...
            freedom
        total_params = self.M + self.N
        start_params = np.zeros(total_params)
        start_params[: self.M] = 0.05
        start_params[self.M :] = 0.05
        bounds = []
        for _i in range(0, total_params):
            bounds.append((0.001, 0.999))
        constraint = {"type": "ineq", "fun": lambda x: 0.999 - x[0] - x[1]}
        res = minimize(
            self.dcc_loglike,
            (start_params),
            args=(data, D_t),
            constraints=constraint,
            bounds=bounds,
            options={"disp": True},
        )
        # possible other option to find a global maximum or minimum
        # res = optimize.shgo(self.dcc_loglike, bounds, args = (data, D_t), ...
            options={'disp':True})
        par_dcc = res.x
        return par_garch_n, par_dcc, D_t

    def dcc_garch_scenarios(self, data: pd.DataFrame, ndays: int, npaths: int) -> Any:
```

```
        """Generate scenarios for universe."""
        data = np.log(np.array(data) + 1)  # set to log returns
        mean_n = data.mean(axis=0)
        self.mean = mean_n
        demean_data = data - mean_n
        demean_data = pd.DataFrame(demean_data)

        par_garch, par_dcc, D_t = self.dcc_fit(demean_data)

        self.par_garch = par_garch
        self.par_dcc = par_dcc
        print(par_garch, par_dcc)

        all_log_returns = np.zeros((npaths, ndays, self.n))
        for s in range(npaths):
            all_log_returns[s] = self.dcc_garch_predict(par_garch, par_dcc, D_t, ...
                demean_data, ndays)

        all_paths, all_log_returnsT = self.cumulative_returns(all_log_returns, ndays, npaths)
        all_returns = np.exp(all_log_returnsT) - 1

        return all_log_returnsT, all_returns, all_paths

    def dcc_garch_predict(
        self,
        par_garch: Any,
        par_dcc: Any,
        D_t: Any,
        demean_data: pd.DataFrame,
        ndays: int,
    ) -> Any:
        """Predict the future return scenarios."""
        a = par_dcc[: self.M]
        b = par_dcc[self.M :]

        lag = max(self.M, self.N)

        data_update = np.array(demean_data)
        Dt1 = D_t
        Q_bar_update = self.Q_bar
        Qt_update = self.Q_tn
        et_update = self.et
        mean_n1 = self.mean
```

```
        returns = np.zeros((ndays, self.n))

        for k in range(ndays):
            # step 1: garch prediction => D_t+1
            ht1 = np.zeros(self.n)

            for i in range(self.n):

                alpha0 = par_garch[i][0]
                alpha = par_garch[i][1 : self.P + 1]
                beta = par_garch[i][self.P + 1 :]

                if self.P == 1:
                    var_alph = alpha * data_update[-1, i] ** 2
                if self.Q == 1:
                    var_bet = beta * Dt1[-1][i]
                else:
                    var_alph = np.dot(alpha, data_update[-1 - self.P : -1, i] ** 2)
                    var_bet = np.dot(beta, Dt1[-1 - self.Q : -1, i])

                ht1[i] = alpha0 + var_alph + var_bet
            Dt1 = np.append(Dt1, [ht1], axis=0)

            # step 2: dcc prediction => R_t+1
            if lag == 1:
                et_i = et_update[:, -1].flatten().reshape((self.n, 1))
                Qt1 = (1.0 - a - b) * Q_bar_update + a * np.matmul(et_i, et_i.T) + b * ...
                    Qt_update[-1]

            else:
                a_sum = np.zeros((self.n, self.n))
                b_sum = np.zeros((self.n, self.n))

                if self.M == 1:
                    a_sum = a * np.matmul(
                        et_update[:, -1].reshape((self.n, 1)),
                        np.transpose(et_update[:, -1].reshape((self.n, 1))),
                    )
                if self.N == 1:
                    b_sum = b * Qt_update[-1]
                else:
                    for m in range(1, self.M):
```

```
                        a_sum = a_sum + a[m - 1] * np.matmul(
                            et_update[:, -1 - m].reshape((self.n, 1)),
                            np.transpose(et_update[:, -1 - m].reshape((self.n, 1))),
                        )
                    for order in range(1, self.N):
                        b_sum = b_sum + b[order - 1] * Qt_update[-order]

                Qt1 = (1 - np.sum(a) - np.sum(b)) * self.Q_bar + a_sum + b_sum

            Q_star = np.diag(np.sqrt(np.diagonal(Qt1)))
            Rt1 = np.matmul(np.matmul(np.linalg.inv(Q_star), Qt1), np.linalg.inv(Q_star))

            # step 3: return calculation => at1 = H_t+1 * z_t+1

            Ht1 = np.matmul(np.diag(Dt1[-1]), np.matmul(Rt1, np.diag(Dt1[-1])))
            zt1 = np.random.default_rng().normal(0, 1, size=(self.n, 1))

            at1 = np.matmul(np.sqrt(Ht1), zt1)
            at1 = at1.flatten()

            # calculate mean_t+k
            mean_n1 = (mean_n1 * (self.T + k) + at1 + mean_n1) / (self.T + k + 1)
            return_k = mean_n1 + at1
            returns[k] = return_k

            # step 4: update of relevant data
            data_update = np.append(data_update, [at1], axis=0)
            et1 = np.matmul(np.linalg.inv(np.diag(Dt1[-1])), np.transpose(data_update[-1, :]))
            et1 = et1.reshape((self.n, 1))
            et_update = np.append(et_update, et1, axis=1)
            # Q_bar_update = (Q_bar_update*(len(data_update)-1) + ...
                np.matmul(et1,et1.T))/(self.T+1)
            Qt_update = np.append(Qt_update, [Qt1], axis=0)

        return returns

    def cumulative_returns(self, all_returns: np.array, ndays: int, scenarios: int) -> Any:
        """Create paths instead of daily returns."""
        real_returns = np.exp(all_returns)
        paths = np.ones((scenarios, self.n, ndays + 1))
        log_returns = np.ones((scenarios, self.n, ndays))
        for s in range(scenarios):
            for k in range(1, ndays + 1):
```

```
                for i in range(self.n):
                    paths[s][i][k] = real_returns[s][k - 1][i]
                    log_returns[s][i][k - 1] = all_returns[s][k - 1][i]
            paths[s] = np.cumprod(paths[s], axis=1)
        return paths, log_returns

    def visualize(
        self,
        paths_per_asset: np.array,
        number_of_assets: int,
        number_of_scenarios: int,
        number_of_days: int,
    ) -> None:
        """Visualize the simulated returns."""
        days = list(range(number_of_days))
        fig, ax = plt.subplots(figsize=(14, 7))
        for i in range(number_of_assets):
            for s in range(number_of_scenarios):
                ax.plot(days, paths_per_asset[i][s], linewidth=2)
        ax.set_xlabel("Time [Days]", fontsize=14)
        ax.set_ylabel("Cummulative Return [/]", fontsize=14)
        ax.set_xlim(0, 19)
        ax.tick_params(axis="both", which="major", labelsize=14)
```

A.2 Block Bootstrap

```
class MovingBlockBootstrap:
    """Class that generates scenarios with Moving Block Bootstrap Method."""

    def __init__(
        self, block_size: int, overlap: int, data: pd.DataFrame, scenarios: int, ndays: int
    ) -> None:
        self.block_size = block_size
        self.overlap = overlap
        self.scenarios = scenarios
        self.ndays = ndays
        self.data = data

    def block_bootstrap(self) -> Any:
        """Create new scenarios by block bootstrapping the original sample."""
        # Otherwise the algorithm cannot work properly
```