S1 Stan code

As supplementary material we provide the full *Stan* code of all models. All implementations adhere to the same interface, taking observed returns as input data. Missing data, i.e. which should not be used during fitting, are indicated with a binary mask. Furthermore, all models compute the log likelihood of all returns. On missing returns these correspond to model predictions.

S1.1 GARCH model

Listing 1 The code for the GARCH model follows the Stan manual (Stan Development Team 2017). The main change being that we allow for missing return data and include model predictions for these. Furthermore, the log likelihood for every return, observed or missing, is computed in the generated quantities block which is automatically run once after each sampling step.

```
data {
  int<lower=0> T;
  int<lower=0, upper=1> miss_mask[T];
  real ret_obs[T]; // Note: Masked indices will be treated as
    missing;
}
transformed data {
  int N = 0; // number of missing values
  for (t in 1:T)
    if (miss_mask[t] == 1) N = N + 1;
}
parameters {
  real mu;
  real<lower=0> alpha0;
  real<lower=0,upper=1> alpha1;
  real<lower=0,upper=(1-alpha1)> beta1;
  real<lower=0> sigma1;
  real eps_miss[N]; // missing normalized return innovations
}
transformed parameters {
  real ret[T]; // returns ... observed or r_t = mu + sigma_t *
    eps_t
  real<lower=0> sigma[T];

  {
    int idx = 1; // missing value index

    sigma[1] = sigma1;
    if (miss_mask[1] == 1) {
      ret[1] = mu + sigma[1] * eps_miss[idx];
      idx = idx + 1;
    } else
      ret[1] = ret_obs[1];

    for (t in 2:T) {
      sigma[t] = sqrt(alpha0
                      + alpha1 * pow(ret[t - 1] - mu, 2)
                      + beta1 * pow(sigma[t - 1], 2));
```

```
      if (miss_mask[t] == 1) {
        ret[t] = mu + sigma[t] * eps_miss[idx];
        idx = idx + 1;
      } else
        ret[t] = ret_obs[t];
    }
  }
}
model {
  mu ~ normal(0, 1);
  sigma1 ~ normal(0, 1);

  ret ~ normal(mu, sigma);
  // Jacobian correction for transformed innovations
  for (t in 1:T) {
    if (miss_mask[t] == 1)
      target += log(sigma[t]);
  }
}
generated quantities {
  real log_lik[T];

  for (t in 1:T)
    log_lik[t] = normal_lpdf(ret_obs[t] | mu, sigma[t]);
}
```

S1.2 SV model

Listing 2 The code for the stochastic volatility model follows the Stan manual (Stan Development Team 2017). As explained there, the latent volatility process is modeled as non-centered.

```
data {
  int<lower=0> T; // time points (equally spaced)
  int<lower=0, upper=1> miss_mask[T];
  vector[T] ret_obs; // Note: Masked indices will be treated as missing;
}
transformed data {
  int N = 0; // number of missing values
  for (t in 1:T)
    if (miss_mask[t] == 1) N = N + 1;
}
parameters {
  real mu_h;                      // mean log volatility
  real<lower=-1,upper=1> phi_h; // persistence of volatility
  real<lower=0> sigma_h;          // white noise shock scale
  vector[T] h_std;                // std log volatility at time t
  vector[N] eps_miss;             // missing normalized return innovations
}
transformed parameters {
  vector[T] h = h_std * sigma_h; // now h ~ normal(0, sigma)
  real ret[T];                    // returns ... observed or r_t = sigma_t * eps_t
  real<lower=0> sigma[T];

  h[1] /= sqrt(1 - phi_h * phi_h); // rescale h[1]
```

```
  h += mu_h;
  sigma[1] = exp(h[1] / 2);

  {
    int idx = 1;

    if (miss_mask[1] == 1) {
      ret[1] = sigma[1] * eps_miss[idx];
      idx = idx + 1;
    } else
      ret[1] = ret_obs[1];

    for (t in 2:T) {
      h[t] += phi_h * (h[t-1] - mu_h);
      sigma[t] = exp(h[t] / 2);

      if (miss_mask[t] == 1) {
        ret[t] = sigma[t] * eps_miss[idx];
        idx = idx + 1;
      } else
        ret[t] = ret_obs[t];
    }
  }
}
model {
  phi_h ~ uniform(-1, 1);
  sigma_h ~ cauchy(0, 5);
  mu_h ~ cauchy(0, 10);
  h_std ~ normal(0, 1);

  ret ~ normal(0, sigma);
  // Jacobian correction for transformed innovations
  for (t in 1:T) {
    if (miss_mask[t] == 1)
      target += h[t] / 2; // = log(sigma[t])
  }
}
generated quantities {
  vector[T] log_lik;

  for (t in 1:T)
    log_lik[t] = normal_lpdf(ret_obs[t] | 0, sigma[t]);
}
```

S1.3 VS model

Listing 3 Stan code for the model by Vikram & Sinha. This model is implemented in two specifications: First, as in the original model, with the fundamental price p_t^* being a running average over past prices.

```
data {
  int<lower=0> T; // time points (equally spaced)
  int<lower=0, upper=1> miss_mask[T];
  vector[T] ret_obs; // Note: Masked indices will be treated as
    missing;
}
transformed data {
  int N = 0; // number of missing values
  real ret_max = max(ret_obs);
  real ret_sd  = sqrt(variance(ret_obs));

  for (t in 1:T)
    if (miss_mask[t] == 1) N = N + 1;
}
parameters {
  real<lower=0> mu;
  real<lower=0> lenscale_raw;
  real<lower=0> sigma_max;
  real log_p_tau_0;
  vector[N] eps_miss; // missing normalized return innovations
}
transformed parameters {
  real ret[T];
  real log_p[T]; // log prices
  real log_p_tau[T];
  real<lower=0> P_b[T]; // Probability of trading P(|S(t)| = 1)
  real<lower=0> sigma[T];
  real<lower=0> lenscale = 1000 * lenscale_raw;
  real<lower=0, upper=1> tau = exp( - 1 / lenscale);

  log_p[1] = 0; // wlog p_0 = 1
  // Moving average of prices, i.e. p_tau[i] = tau * p_tau[i-1] +
    (1 - tau) * p[i]
  log_p_tau[1] = log_mix(tau, log_p_tau_0, log_p[1]);
  P_b[1] = exp(- mu * fabs(log_p[1] - log_p_tau[1]));
  sigma[1] = sigma_max * sqrt(2 * P_b[1]);
  {
    int idx = 1;

    if (miss_mask[1] == 1) {
      ret[1] = sigma[1] * eps_miss[idx];
      idx = idx + 1;
    } else
      ret[1] = ret_obs[1];

    for (t in 2:T) {
      // Note: index shift between prices and returns
      log_p[t] = log_p[t - 1] + ret[t - 1];
      log_p_tau[t] = log_mix(tau, log_p_tau[t - 1], log_p[t]);
      P_b[t] = exp(- mu * fabs(log_p[t] - log_p_tau[t - 1]));
      sigma[t] = sigma_max * sqrt(2 * P_b[t]);
```

```
      if (miss_mask[t] == 1) {
        ret[t] = sigma[t] * eps_miss[idx];
        idx = idx + 1;
      } else
        ret[t] = ret_obs[t];

    }
  }
}
model {
  mu ~ gamma(3, 0.03);
  lenscale_raw ~ inv_gamma(2, 1); // avoid lower boundary ...
    lenscale ~ inv_gamma(2, 1000)
  log_p_tau_0 ~ normal(0, 0.1);
  sigma_max ~ normal(ret_max, ret_max / 4);

  ret ~ normal(0, sigma);
  // Jacobian correction for transformed innovations
  for (t in 1:T) {
    if (miss_mask[t] == 1)
      target += log(sigma[t]);
  }
}
generated quantities {
  vector[T] log_lik;

  for (t in 1:T)
    log_lik[t] = normal_lpdf(ret_obs[t] | 0, sigma[t]);
}
```

Listing 4 Secondly, with the fundamental log price $\log p_t^*$ following a Brownian motion. As explained in the text, in order to increase sampling efficiency this random walk is implemented in non-centered parameters, i.e. innovations.

```
data {
  int<lower=0> T; // time points (equally spaced)
  int<lower=0, upper=1> miss_mask[T];
  vector[T] ret_obs; // Note: Masked indices will be treated as
    missing;
}
transformed data {
  int N = 0; // number of missing values
  real ret_max = max(ret_obs);
  real ret_sd  = sqrt(variance(ret_obs));

  for (t in 1:T)
    if (miss_mask[t] == 1) N = N + 1;
}
parameters {
  real<lower=0> mu;
  real<lower=0> sigma_max;
  real log_p_tau_0;
  vector[N] eps_miss; // missing normalized return innovations
  // random walk for fundamental price
  vector[T] log_p_tau_raw;
  real<lower=0> sigma_p_tau;
}
```

```
transformed parameters {
  real ret[T];
  real log_p[T]; // log prices
  real log_p_tau[T];
  real<lower=0> P_b[T]; // Probability of trading P(|S(t)| = 1)
  real<lower=0> sigma[T];

  log_p[1] = 0; // wlog p_0 = 1
  // Random walk of fundamental price
  log_p_tau[1] = log_p_tau_0 + sigma_p_tau * log_p_tau_raw[1];
  P_b[1] = exp(- mu * fabs(log_p[1] - log_p_tau[1]));
  sigma[1] = sigma_max * sqrt(2 * P_b[1]);
  {
    int idx = 1;

    if (miss_mask[1] == 1) {
      ret[1] = sigma[1] * eps_miss[idx];
      idx = idx + 1;
    } else
      ret[1] = ret_obs[1];

    for (t in 2:T) {
      // Note: index shift between prices and returns
      log_p[t] = log_p[t - 1] + ret[t - 1];
      log_p_tau[t] = log_p_tau[t - 1] + sigma_p_tau *
    log_p_tau_raw[t];
      P_b[t] = exp(- mu * fabs(log_p[t] - log_p_tau[t - 1]));
      sigma[t] = sigma_max * sqrt(2 * P_b[t]);

      if (miss_mask[t] == 1) {
        ret[t] = sigma[t] * eps_miss[idx];
        idx = idx + 1;
      } else
        ret[t] = ret_obs[t];

    }
  }
}
model {
  mu ~ gamma(3, 0.03);
  log_p_tau_0 ~ normal(0, 0.1);
  log_p_tau_raw ~ normal(0, 1);
  sigma_max ~ normal(ret_max, ret_max / 4);

  ret ~ normal(0, sigma);
  // Jacobian correction for transformed innovations
  for (t in 1:T) {
    if (miss_mask[t] == 1)
      target += log(sigma[t]);
  }
}
generated quantities {
  vector[T] log_lik;

  for (t in 1:T)
    log_lik[t] = normal_lpdf(ret_obs[t] | 0, sigma[t]);
}
```

S1.4 FW model

Listing 5 Stan code for the model by Franke & Westerhoff. Agent dynamics follows the DCA-HPM specification and again two specifications are assumed for the fundamental price dynamics. Here, the fundamental price is computed as a running average over past price. Note that as in the original model, the *log* fundamental price is denoted by p_t^*.

```
data {
  int<lower=0> T; // time points (equally spaced)
  int<lower=0, upper=1> miss_mask[T];
  vector[T] ret_obs; // Note: Masked indices will be treated as
    missing;
}
transformed data {
  int N = 0; // number of missing values
  real ret_sd  = sqrt(variance(ret_obs));
  // mu and beta fixed ... redundant anyways
  real mu = 0.01;
  real beta = 1.0;

  for (t in 1:T)
    if (miss_mask[t] == 1) N = N + 1;
}
parameters {
  real<lower=0> phi;
  real<lower=0> xi;
  real alpha_0;
  real<lower=0> alpha_n;
  real<lower=0> alpha_p;
  real<lower=0> sigma_f;
  real<lower=0> sigma_c;
  real<lower=0, upper=1> n_f_1;
  real<lower=0> lenscale_raw;
  real p_star_0;
  vector[N] eps_miss; // missing normalized return innovations
}
transformed parameters {
  vector[T] n_f;
  vector[T] demand;
  vector[T] sigma;
  // Note: All prices are actually log prices!
  vector[T] p_star;
  vector[T] p;
  real ret[T];
  real<lower=0> lenscale = 1000 * lenscale_raw;
  real<lower=0, upper=1> tau = exp( - 1 / lenscale);

  p[1] = 0; // wlog log p_1 = 0
  p_star[1] = log_mix(tau, p_star_0, p[1]);

  n_f[1] = n_f_1;
  demand[1] = 0;
  sigma[1] = mu * sqrt( square(n_f[1] * sigma_f)
      + square((1 - n_f[1]) * sigma_c));
  {
    int idx = 1;

```

```
    if (miss_mask[1] == 1) {
      ret[1] = sigma[1] * eps_miss[idx];
      idx = idx + 1;
    } else
      ret[1] = ret_obs[1];

    for (t in 2:T) {
      // Note: index shift between prices and returns
      p[t] = p[t - 1] + ret[t - 1];
      p_star[t] = log_mix(tau, p_star[t-1], p[t]);

      {
  // equation (HPM)
  real a = alpha_n * (n_f[t-1] - (1 - n_f[t-1]))
    + alpha_0
    + alpha_p * square(p[t-1] - p_star[t-1]);
  // equation (DCA)
  n_f[t] = inv_logit(beta * a);
  demand[t] = mu * (n_f[t] * phi * (p_star[t] - p[t])
        + (1 - n_f[t]) * xi * (p[t] - p[t-1]));
  // structured stochastic volatility
  sigma[t] = mu * sqrt( square(n_f[t] * sigma_f)
             + square((1 - n_f[t]) * sigma_c));
      }

      if (miss_mask[t] == 1) {
        ret[t] = sigma[t] * eps_miss[idx];
        idx = idx + 1;
      } else
        ret[t] = ret_obs[t];
    }
  }
}
model {
  phi ~ student_t(5, 0, 1);
  xi ~ student_t(5, 0, 1);
  alpha_0 ~ student_t(5, 0, 1);
  alpha_n ~ student_t(5, 0, 1);
  alpha_p ~ student_t(5, 0, 1);
  sigma_f ~ normal(0, ret_sd / mu);
  sigma_c ~ normal(0, 2.0 * ret_sd / mu);
  p_star_0 ~ normal(0, 0.2);
  lenscale_raw ~ inv_gamma(2, 1); // avoid lower boundary ...
    lenscale ~ inv_gamma(2, 1000)

  // Price likelihood
  ret ~ normal(demand, sigma);
  // Jacobian correction for transformed innovations
  for (t in 1:T) {
    if (miss_mask[t] == 1)
      target += log(sigma[t]);
  }
}
generated quantities {
  vector[T] log_lik;

  for (t in 1:T)
    log_lik[t] = normal_lpdf(ret_obs[t] | 0, sigma[t]);
```

```
}
```

Listing 6 Stan code for the model by Franke & Westerhoff with the log fundamental price p_t^* following a random walk.

```
data {
  int<lower=0> T; // time points (equally spaced)
  int<lower=0, upper=1> miss_mask[T];
  vector[T] ret_obs; // Note: Masked indices will be treated as missing;
}
transformed data {
  int N = 0; // number of missing values
  real ret_sd  = sqrt(variance(ret_obs));
  // mu and beta fixed ... redundant anyways
  real mu = 0.01;
  real beta = 1.0;

  for (t in 1:T)
    if (miss_mask[t] == 1) N = N + 1;
}
parameters {
  real<lower=0> phi;
  real<lower=0> xi;
  real alpha_0;
  real<lower=0> alpha_n;
  real<lower=0> alpha_p;
  real<lower=0> sigma_f;
  real<lower=0> sigma_c;
  real<lower=0, upper=1> n_f_1;
  // p_star random walk in non-centered parameterization
  vector[T] epsilon_star;
  real<lower=0> sigma_p_star;
  vector[N] eps_miss; // missing normalized return innovations
}
transformed parameters {
  vector[T] n_f;
  vector[T] demand;
  vector[T] sigma;
  // Note: All prices are actually log prices!
  vector[T] p_star;
  vector[T] p;
  real ret[T];

  p[1] = 0; // wlog log p_1 = 0
  p_star[1] = p[1] + epsilon_star[1]; // fixme ... interpretation epsilon_raw[1]

  n_f[1] = n_f_1;
  demand[1] = 0;
  sigma[1] = mu * sqrt( square(n_f[1] * sigma_f)
      + square((1 - n_f[1]) * sigma_c));
  {
    int idx = 1;

    if (miss_mask[1] == 1) {
      ret[1] = sigma[1] * eps_miss[idx];
      idx = idx + 1;
```

```
    } else
      ret[1] = ret_obs[1];

    for (t in 2:T) {
      // Note: index shift between prices and returns
      p[t] = p[t - 1] + ret[t - 1];
      p_star[t] = p_star[t-1] + sigma_p_star * epsilon_star[t];

      {
  // equation (HPM)
  real a = alpha_n * (n_f[t-1] - (1 - n_f[t-1]))
    + alpha_0
    + alpha_p * square(p[t-1] - p_star[t-1]);
  // equation (DCA)
  n_f[t] = inv_logit(beta * a);
  demand[t] = mu * (n_f[t] * phi * (p_star[t] - p[t])
        + (1 - n_f[t]) * xi * (p[t] - p[t-1]));
  // structured stochastic volatility
  sigma[t] = mu * sqrt( square(n_f[t] * sigma_f)
             + square((1 - n_f[t]) * sigma_c));
      }

      if (miss_mask[t] == 1) {
        ret[t] = sigma[t] * eps_miss[idx];
        idx = idx + 1;
      } else
        ret[t] = ret_obs[t];
    }
  }
}
model {
  phi ~ student_t(5, 0, 1);
  xi ~ student_t(5, 0, 1);
  alpha_0 ~ student_t(5, 0, 1);
  alpha_n ~ student_t(5, 0, 1);
  alpha_p ~ student_t(5, 0, 1);
  sigma_f ~ normal(0, ret_sd / mu);
  sigma_c ~ normal(0, 2.0 * ret_sd / mu);
  epsilon_star ~ normal(0, 1);
  sigma_p_star ~ normal(0, ret_sd / 2.0);

  // Price likelihood
  ret ~ normal(demand, sigma);
  // Jacobian correction for transformed innovations
  for (t in 1:T) {
    if (miss_mask[t] == 1)
      target += log(sigma[t]);
  }
}
generated quantities {
  vector[T] log_lik;

  for (t in 1:T)
    log_lik[t] = normal_lpdf(ret_obs[t] | 0, sigma[t]);
}
```

S1.5 ALW model

Listing 7 Stan code for the model by Alfarano, Lux & Wagner.

```
data {
  int<lower=0> T; // time points (equally spaced)
  int<lower=0, upper=1> miss_mask[T];
  vector[T] ret_obs; // Note: Masked indices will be treated as missing;
}
transformed data {
  int N = 0; // number of missing values
  real T_quot = 1; // wlog fixed at 1

  for (t in 1:T)
    if (miss_mask[t] == 1) N = N + 1;
}
parameters {
  vector<lower=-1, upper=1>[T + 1] x; // sentiment over time
  real<lower=0> sigma_f;
  real<lower=0> alpha;
  real<lower=0> beta;
  vector[N] eps_miss; // missing normalized return innovations
}
transformed parameters {
  vector[T] ret;

  {
    int idx = 1;

    for (t in 1:T) {
      if (miss_mask[t] == 1) {
        ret[t] = T_quot * (x[t + 1] - x[t]) + sigma_f * eps_miss[idx];
        idx = idx + 1;
      } else
        ret[t] = ret_obs[t];
    }
  }
}
model {
  sigma_f ~ normal(0, 1);
  alpha ~ normal(0, 1);
  beta ~ normal(0, 1);

  // x[1] implicitly uniform
  for (t in 2:(T+1))
    x[t] ~ normal(x[t - 1] - 2.0 * alpha * x[t - 1],
                  sqrt(2.0 * beta * (1 - square(x[t - 1]))));

  // Price likelihood
  ret ~ normal(T_quot * (x[2:(T+1)] - x[1:T]), sigma_f);
  // Jacobian correction for transformed innovations
  for (t in 1:T) {
    if (miss_mask[t] == 1)
      target += log(sigma_f);
  }
```

```
}
generated quantities {
  vector[T] sigma;
  vector[T] log_lik;

  for (t in 1:T) {
    sigma[t] = sqrt(square(T_quot * (x[t + 1] - x[t])) + square(
    sigma_f));
    log_lik[t] = normal_lpdf(ret_obs[t] | T_quot * (x[t + 1] - x[t
    ]), sigma_f);
  }
}
```