Simulating tree species migration: from post-glacial to future climate change

Heike Lischke[1], Veiko Lehsten[1,2], and Deborah Zani[1,2]

[1]Swiss Federal Institute of Forest, Snow and Landscape Research WSL
[2]University of Lund, Sweden

This Notebook reproduces the analyses of the paper Zani et al. Geosci. Model Dev., XX, 20XX, including the generation of figures and supplement tables. It also allows to inspect the data in more details.

Abstract

The prediction of species geographic redistribution under climate change (i.e. range shifts) has been addressed by both experimental and modelling approaches and can be used to inform efficient policy measures on the functioning and services of future ecosystems. Dynamic Global Vegetation Models (DGVMs) are considered state-of-the art tools to understand and quantify the spatio-temporal dynamics of ecosystems at large scales and their response to changing environments. They can explicitly include local vegetation dynamics relevant to migration (establishment, growth, seed production), species-specific dispersal abilities and the competitive interactions with other species in the new environment. However, the inclusion of more detailed mechanistic formulations of range shift processes may also widen the overall uncertainty of the model. Thus, a quantification of these uncertainties is needed to evaluate and improve our confidence in the model predictions. In this study, we present an efficient assessment of parameter and model uncertainties combining low-cost analyses in successive steps: local sensitivity analysis, exploration of the performance landscape at extreme parameter values, and inclusion of relevant ecological processes in the model structure. This approach was tested on the newly-implemented migration module of the state-of-the-art DGVM, LPJ-GM. Estimates of post-glacial migration rates obtained from pollen and macrofossil records of dominant European tree taxa were used to test the model performance. The results indicate higher sensitivity of migration rates to parameters associated with the dispersal kernel (dispersal distances and kernel shape) compared to plant traits (germination rate and maximum fecundity) and highlight the importance of representing rare long-distance dispersal events via fat-tailed kernels. Overall, the successful parametrization and model selection of LPJ-GM will allow simulating plant migration with a more mechanistic approach at larger spatial and temporal scales, thus improving our efforts to understand past vegetation dynamics and predict future range shifts in a context of global change.

1 How to run this notebook

After downloading the Supplement folder containing this notebook here, activate the environment `environment.yml` and start the notebook from the command line:

```
(base)%pwd >conda activate LPJGM_uncertainty
```

```
(LPJGM_uncertainty)%pwd >jupyter-notebook
```

Input data are included in the folder at the address `%pwd`.

[1]:

```
datapath = %pwd
```

2 Libraries

[2]:

```
import numpy as np
from numpy import nanmean
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import statistics
from scipy import stats
from scipy.stats import mannwhitneyu
from statsmodels.sandbox.stats.multicomp import multipletests
from statsmodels.stats.multicomp import pairwise_tukeyhsd
import copy
from sklearn import preprocessing
from sklearn.preprocessing import StandardScaler
from scipy.special import gamma
```

3 Observational data: estimates of past migration rates

We used estimates of migration rates from paleo-records of European forest expansion after the Last Glacial Maximum (LGM) for the parameter optimization of 17 major European tree species implemented in LPJ-GM. Upper and lower boundaries for the range values of migration rates were derived from different studies based on the method employed for their estimation.

Table 1. Estimates of maximum post-glacial migration rates in meters year^{-1}, dispersal syndromes, and expected competition during post-glacial expansion for 17 major European tree species. Lower and upper boundaries of migration rates correspond to `OBS_MIN` and `OBS_MAX`, respectively, in `Input_EVA.csv` and `Input_KA.csv`.

Species (LPJ-GM notation)	Migration rates	Dispersal syndrome(e)	Competitor(f)
Abies alba (Abi_alb)	250(a) – 300(b)	W	Bet_pen & C3G
Betula pendula (Bet_pen)	540(c) – 800(a)	W, Wa, B	Bet_pub & C3G
Betula pubescens (Bet_pub)	540(c) – 800(a)	W, Wa, B	Bet_pen & C3G
Carpinus betulus (Car_bet)	500(a) – 1000(b)	W, Wa, B, SA	Bet_pen & C3G
Corylus avellana (Cor_ave)	1000(a) – 1500(b)	B, SA	Bet_pen & C3G
Fagus sylvatica (Fag_syl)	200(b) – 300(b)	SA, LA	Bet_pen & C3G
Fraxinus excelsior (Fra_exc)	200(b) – 500(b)	W, Wa, B, LA	Bet_pen & C3G
Picea abies (Pic_abi)	150(d) – 500(b)	W, Wa, B, LA	Bet_pen & C3G
Picea sitchensis (Pic_sit)	150(d) – 500(b)	W	Bet_pen & C3G
Pinus sylvestris (Pin_syl)	600(a) – 1500(b)	W, Wa	Bet_pen & C3G
Pinus halepensis (Pin_syl)	600(a) – 1500(b)	W	Bet_pen & C3G

Species (LPJ-GM notation)	Migration rates	Dispersal syndrome(e)	Competitor(f)
Quercus coccifera (Que_coc)	300(d) – 500(b)	SA	Bet_pen & C3G
Quercus ilex (Que_ile)	300(d) – 500(b)	SA	Bet_pen & C3G
Quercus pubescens (Que_pub)	300(d) – 500(b)	SA	Bet_pen & C3G
Quercus robur (Que_rob)	300(d) – 500(b)	SA	Bet_pen & C3G
Tilia cordata (Til_cor)	150(b) – 500(b)	SA	Bet_pen & C3G
Ulmus glabra (Ulm_gla)	550(d) – 1000(b)	W, Wa	Bet_pen & C3G

(a) Estimates of maximum migration rates by Giesecke and Brewer (2018) with pollen analysis corrected by phylogeographic studies

(b) Estimates of maximum migration rates by Huntley and Birks (1983) with pollen analysis

(c) Estimates of maximum migration rates by Feurdean et al. (2013) with fossil records, assuming spread from southern refugia (40-45ºN latitude)

(d) Estimates of overall migration rates by Giesecke et al. (2017) derived from the increase in area of presence from interpolated pollen maps and threshold values for pollen presence

(e) Dispersal syndromes as reported by Vittoz and Engler (2007) and TRY Database (Kattge et al., 2011): W = wind; Wa = water; B = bird; LA = large mammal (deer, badger, cattle); SA = small animal (e.g. hoarding by rodents)

(f) Competitor in model simulations: Bet_pen / Bet_pub = Betula pendula / Betula pubescens; C3G = boreal/temperate grasses (C3 photosynthesis pathway).

4 Model: LPJ-GM

LPJ-GM (Lehsten et al., 2019) couples a dynamic migration module to the widely-used DGVM, LPJ-GUESS (where LPJ-GM is short for LPJ-GUESS-MIGRATION). LPJ-GUESS employs a gap model approach for the simulation of ecophysiological processes and the structural dynamics of forests, including species composition and vertical and horizontal heterogeneity (Smith et al. 2001). Thus, the migration module of LPJ-GM simulates local vegetation dynamics and allows species to disperse simultaneously while interacting with each other. Following the TreeMig model implementation (Lischke and Löffler (2006)), LPJ-GM simulates migration at an yearly time-step through four main processes: - **seed production**. The number of seeds produced S is the product of maximum fecundity (FEC_{max}) and the proportion of current leaf area (LAI_{ind}) to the maximum (LAI_{max}):

$$S = FEC_{max} \times \frac{LAI_{ind}}{LAI_{max}}$$

- **seed dispersal**. The seeds $S(x', y')$ produced at a location (x', y') then are distributed according to a probability density function (pdf), i.e. the seed dispersal kernel k_s, so that the seed input $S_d\ (x, y)$ in location x, y is obtained by integrating over all other locations x', y'. The dispersal kernel k_s is a linear combination of two pdfs for short- (SDD) and long-distance dispersal (LDD), where LDD_p is the proportion of LDD, SDD_d and LDD_d are the average distances for SDD and LDD, respectively. In the default setting of LPJ-GM, the pdfs for both SDD and LDD components are negative exponentials.

$$S_d(x,y) = \int S(x',y') \times k_s(x-x',y-y')dx'dy'$$

$$k_s = (1 - LDD_p) \times pdf(z, SDD_d) + LDD_p \times pdf(z, LDD_d)$$

- **seed bank dynamics**. Seed bank dynamics are defined by the yearly change of dormant seeds in the soil S_{sb} that can germinate in the following years. S_{sb} increases by the seed input S_d and decreases by the number of germinated seeds (where $GERM_p$ is the rate of germination), or by loss of seeds ($_s = 0.8$ as annual rate):

$$S_{sb,t+1} = S_{sb,t+1} \times (1 - GERM_p) \times (1 - {}_s)$$

$$S_{sb,t+1} = S_{sb,t} + S_{d,t+1}$$

- **seedling establishment**. The probability of seedling establishment in a certain year EST_p depends on the number of available seeds for germination (S_{sb}) and on the germination rate. The established seedlings grow, compete, and die according to the LPJ-GUESS approach. After reaching maturity, established saplings start producing seeds according the S formula.

$$EST_p = 0.01 \times S_{sb} \times GERM_p$$

At the end of the simulation, the species-specific migration rate (meters year^{-1}) is calculated as the migration distance divided by migration time, i.e. simulation time elapsed since when trees are allowed to perform seed dispersal. Migration distance is obtained by the direct output of LPJ-GM, leaf area index (LAI), as the distance between the starting point of migration and the 95th percentile farthest point in the terrain where LAI exceeds 0.5.

4.1 Simulation settings

Simulations were performed for a total of 500 years and over an area of 201 x 201 cells with corridors each 200 km (for a total of 1,197 grid cells), where tree species were allowed to establish freely in the upper-left corner of the simulation landscape (the starting point of migration). We applied a static suitable climate for all species and an entirely permeable terrain to all grid cells and across all simulation years in order to reproduce optimal environmental conditions for each species' spread. We used the Fast Fourier transform method (FFTM) to enhance the computational efficiency of seed dispersal, as described in Lehsten et al., 2019.

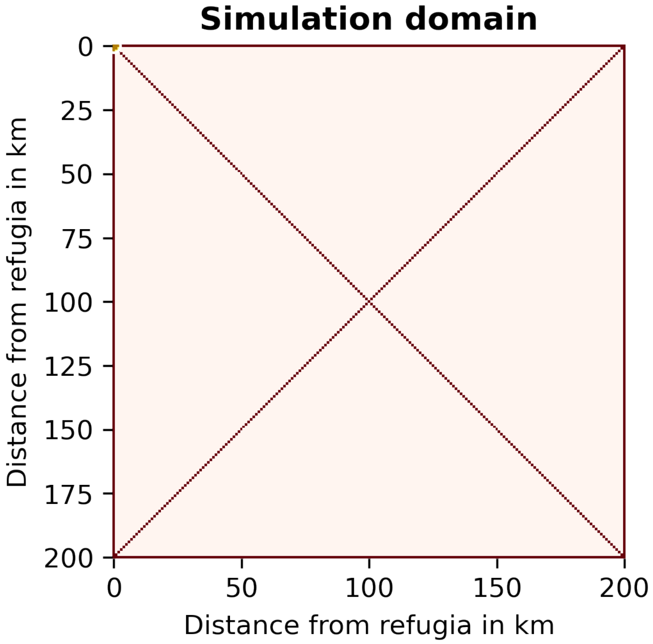
[3]:

```
## Simulation domain
f, ax = plt.subplots()
f.set_figheight(3.54)
f.set_figwidth(3.54)
f.dpi = 300

# Initialize the spatial domain (201x2001 grid cells)
sim_domain = np.zeros((201, 201), int)
```

Parameter	Value	Description
vegmode	“cohort”	simulation mode, either “cohort”, “individual” or “population”
nyear_spinup	53	number of years to spin up the simulation for
ifcalcsla	1	whether to calculate SLA from leaf longevity
ifcalccton	1	whether to calculate leaf C:N min from leaf longevity
firemodel	“NOFIRE”	whether to implement fire (BLAZE, GLOBFIRM) or not (NOFIRE)
weathergenerator	“GWGEN”	whether to generate climate at a smaller scale or interpolate it
npatch	1	number of replicate patches to simulate
npatch_secondarystand	1	number of replicate patches to simulate in secondary stands
reduce_all_stands	0	whether to reduce equal percentage of all stands of a stand type at land cover change
age_limit_reduce	5	Minimum age of stands to reduce at land cover change
patcharea	1000	patch area (m2)
estinterval	5	years between establishment events in cohort mode
ifdisturb	1	whether generic patch-destroying disturbances enabled
distinterval	400	average return time for generic patch-destroying disturbances
ifbgestab	1	whether background establishment enabled
ifsme	1	whether spatial mass effect enabled
ifstochestab	0	whether establishment stochastic
ifstochmort	0	whether mortality stochastic
ifcdebt	1	whether to allow vegetation C storage (1) or not (0)
wateruptake	“rootdist”	“wcont”, “rootdist”, “smart” or “speciesspecific”
rootdistribution	“jackson”	how to parameterise root distribution. Alternatives are “fixed” or “jackson”
textured_soil	1	whether to use silt/sand fractions specific to soiltype
ifcentury	1	whether to use CENTURY SOM dynamics (mandatory for N cycling)
ifnlim	1	whether plant growth limited by available N
freenyears	100	number of years to spin up without N limitation (needed to build up a N pool)
nfix_a	0.102	first term in N fixation eqn (Conservative 0.102, Central 0.234, Upper 0.367)
nfix_b	0.524	second term in N fixation eqn (Conservative 0.524, Central -0.172, Upper -0.754)
nrelocfrac	0.5	fraction of N retranslocated prior to leaf and root shedding
ifsmoothgreffmort	1	whether to vary mort_greff smoothly with growth efficiency (1) or to use the standard step-function (0)
ifdroughtlimitedestab	0	whether establishment is limited by growing season drought
ifrainonwetdaysonly	1	whether to rain on wet days only (1), or to rain a bit every day (0)
ifbvoc	0	whether to include BVOC calculations (1) or not (0)
searchradius_soil	0.01	search for soil data in up to X degrees around the EMDI coordinates

Parameter	Value	Description
years_total	500	total number of simulation years
domain	11, 49, 0.01, 0.01	coordindates of north-west corner and longitude, latitude increments
subdomains	201, 201, 1, 1	size of the domain and number of subdomains in lat-/long-itudinal direction
dispersal_patchsize	0.99	patch size for dispersal calculations (km2)
dispersal	“fft”	whether to use the Fast Fourier transform method for seed dispersal
stochastic_seed_est_scaler	0.01	scaler for stochastic seedling establishment from dispersed seed
log_stochastic_seed_est_scaler	0	if larger than 0 no log stochastic distribution for stochastic seedling establishment from dispersed seeds
output_interval	1	number of years between each output-year should be larger or equal to 1
EDGE_CELLS	40	number of gridcells to fold into the central array of dispersed seeds to avoid a torus effect

```python
# Add corridors as major diagonals and perimeter
np.fill_diagonal(sim_domain, 1)
np.fill_diagonal(np.fliplr(sim_domain), 1)
sim_domain[0,:] = 1
sim_domain[:,0] = 1
sim_domain[200,:] = 1
sim_domain[:,200] = 1

# Plot
ax.imshow(sim_domain, cmap='Reds', interpolation='nearest')

# Add refugium
ax.scatter(0,0, s=25, edgecolor='white', facecolor=(1, 1, 0, 0.5))

ax.set_frame_on(False)
ax.set_ylabel('Distance from refugia in km')
ax.set_xlabel('Distance from refugia in km')
ax.set_title('Simulation domain',fontweight='bold')

# Export figure
plt.savefig('Figure_S1.png', bbox_inches='tight', dpi=300)
plt.show()
```



5 Migration parameters

Since the underlying framework, LPJ-GUESS, has been already extensively validated for different metrics of vegetation composition and structure (e.g. Pappas et al. 2013), we focused our calibration effort on the newly-added migration parameters: FEC_{max}, $GERM_{p}$, SDD_{d} and LDD_{d}. Range values of each parameter were compiled by reviewing research articles and public databases as indicated in **Table 2**. We identified the minimum and maximum values per parameter as the lower and upper bounds, respectively, and calculated the mean along with the 25^{th} and 75^{th} percentiles, assuming a normal distribution (see `Table B1` in the main text and `SA_input` in this notebook for species-specific values: `Default`,`Min`,`25th`,`Mean`,`75th`,`Max`, where default parameters correspond to the values reported by Lischke and Löffler (2006)). `var%` indicates the normalized variability of species-specific parameters, in percentage: $\frac{x_{max}-x_{min}}{x_{max}} \times 100$.

Table 2. Description of the migration parameters and data source: a = Lischke and Löffler (2006); b = TRY database (Kattge et al., 2011); c = Royal Botanic Gardens Kew Seed Information Database (SID); d = Vittoz and Engler (2007); e = Tamme et al. (2014).

Parameter	Notation	Unit	References
Maximum fecundity per tree and year	FEC_max	no.seeds (in 100)	a, b, c
Seed germination rate	GERM_p	%	a, b, c
Average short dispersal distance	SDD_d	meters	a, b, d
Average long dispersal distance (1%)	LDD_d	meters	a, b, d, e

6 Helper functions

[4]:

```
# Scale values from 0 to 1
def scaler_0_1(df):
    return (df - np.min(df)) / (np.max(df) - np.min(df))

# Mean normalization
# i.e. centers around 0 while keeping positive and negative values
def mean_norm(data):
    return (data - np.mean(data)) / (max(data) - min(data))

# Calculate RMSE
def rmse_perc(sim,obs,obs_mean):
    MSE = np.square(np.subtract(sim,obs)).mean()
    RMSE = np.sqrt(MSE)
    return (100 / obs_mean) * RMSE

# Calculate standard error
f_ste = lambda x : x.std() / np.sqrt( x.count() )
f_ste.__name__ = 'ste'

# Find minimum residuals
def min_residual(group) :
    residuals = group.residuals_abs
    output = group[residuals == residuals.min()]
    if (output.shape[0] > 1) :
        output = output.head(1)
    return output
```

7 Load data

[5]:

```
# Load data (\\ for Windows)
SA_input = pd.read_table(datapath+'\\Input_SA.csv',header=0,delimiter=';')
EVA_input = pd.read_table(datapath+'\\Input_EVA.csv',header=0,delimiter=';')
KA_input = pd.read_table(datapath+'\\Input_KA.csv',header=0,delimiter=';')
SA_fatkernels_input = pd.read_table(datapath+'\\Input_SA_model2.
 ↪csv',header=0,delimiter=';')
```

[6]:

```
SA_input.iloc[:,0:10].to_csv('Table_B1.csv', sep=';', index=False)
```

8 Evaluation of parameter uncertainty

8.1 Local Sensitivity Analysis (LSA)

As a first step, we applied a species-specific local sensitivity analysis (LSA). We followed the approach by Downing et al. (1985) and applied a 5-points one-at-a-time sensitivity analysis, where one parameter is adjusted to its minimum, mean, maximum, 25^{th} and 75^{th} percentile values, while the others are kept at their default values (`Table S1`). We quantified the response of the model to each parameter in terms of directionality, linearity and magnitude by four summary statistics: the Sensitivity Index (SI), two Importance Indexes (II1 and II2), and the Linearity Index (LI) (Downing et al. (1985); Hamby, 1995):

$$SI = \frac{\Delta y}{\Delta x_i}$$

where Δy is the average of the differences of the output values for the 5-points, and Δx_i is the corresponding 25% value change for each input parameter x_i.

The first Importance Index $II_{1,i}$ of parameter i is the product between the uncertainty of parameter i (UI_i) and its effect on the model ouput SI_i:

$$II_{1,i} = UI_i \times SI_i$$

$$UI = \frac{x_{max} - x_{min}}{x_{max}}$$

where the uncertainty UI_i of parameter i is its scaled range from its minimum to its maximum.

The second Importance Index $II_{2,i}$ of parameter i is calculated as the percentage difference of the output y (i.e. migration rate) when varying the input parameter x_i from its minimum to its maximum:

$$II_{2,i} = \frac{y_{max} - y_{min}}{y_{max}}$$

The Linearity Index LI_i indicates whether the relationship between each input parameter and the model output approach linearity:

$$LI_i = \frac{y_{max} - y_{min}}{\sqrt{s^2}}$$

where s^2 is the sample variance of the model output and an exact linear relationship between model output and parameter corresponds to $LI_i = 1$.

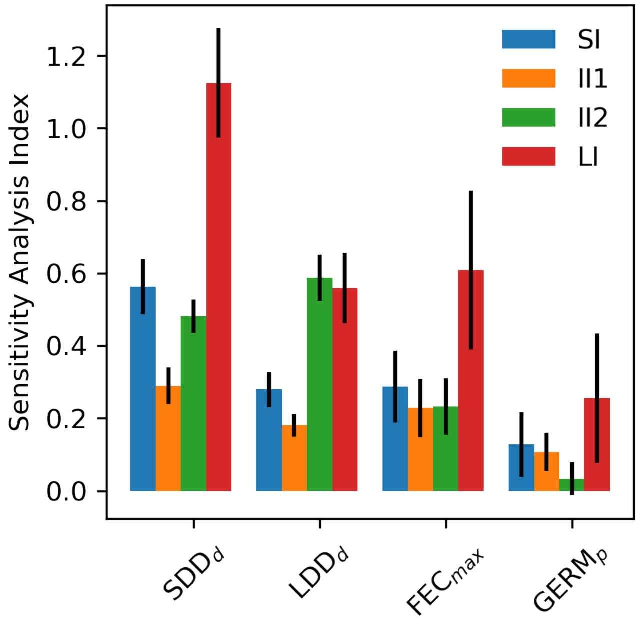
[7]:

```
# Calculate parameter uncertainty (UI)
SA_input['UI'] = SA_input['Max'] - SA_input['Min']

# Print UI statistics
print('Parameter uncertainty mean:')
print(SA_input[['Parameters','UI']].groupby('Parameters').mean())
print('Parameter uncertainty standard deviation:')
print(SA_input[['Parameters','UI']].groupby('Parameters').std())

# Calculate Sensitivity Index (SI)
SA_input['X1'] = SA_input['25th'] - SA_input['Min']
SA_input['X2'] = SA_input['Mean'] - SA_input['25th']
SA_input['X3'] = SA_input['75th'] - SA_input['Mean']
SA_input['X4'] = SA_input['Max'] - SA_input['75th']
SA_input['Y1'] = SA_input['MigRate_25th'] - SA_input['MigRate_Min']
SA_input['Y2'] = SA_input['MigRate_Mean'] - SA_input['MigRate_25th']
SA_input['Y3'] = SA_input['MigRate_75th'] - SA_input['MigRate_Mean']
SA_input['Y4'] = SA_input['MigRate_Max'] - SA_input['MigRate_75th']
SA_input['SI1'] = SA_input['Y1'] / SA_input['X1']
SA_input['SI2'] = SA_input['Y2'] / SA_input['X2']
SA_input['SI3'] = SA_input['Y3'] / SA_input['X3']
SA_input['SI4'] = SA_input['Y4'] / SA_input['X4']
SA_input['SI'] = SA_input[['SI1','SI2','SI3','SI4']].mean(axis=1)

# Calculate Importance Index (II1)
# product of sensitivity and normalized [0-1] parameter uncertainty
UI_norm = (SA_input['Max'] - SA_input['Min']) / SA_input['Max']
SA_input['II1'] = UI_norm * SA_input['SI']

# Calculate Importance Index (II2)
# output percentage difference
SA_input['II2'] = (SA_input['MigRate_Max'] - SA_input['MigRate_Min']) /␣
 ↪SA_input['MigRate_Max']

# Calculate the Linearity Index (LI)
# ratio between output range and ca. standard deviation of parameter range
SA_input['LI'] = (SA_input['MigRate_Max'] - SA_input['MigRate_Min']) /␣
 ↪((SA_input['Max'] - SA_input['Min']) / 2)

# Calculate summary statistic
SA_input['SA_total'] = SA_input['SI'] + SA_input['II1'] + SA_input['II2'] +␣
 ↪SA_input['LI']

# Export Supplement Table 1
Supplement_Table_S1 =␣
 ↪SA_input[['Species','Parameters','UI','SI','II1','II2','LI']]

# Save table
Supplement_Table_S1.to_csv('Table_S1.csv', sep=';', index=False)

# Summary statistics
Supplement_Table_S1.groupby('Parameters').describe()
```

```
# Plot Figure 1
SI_df = SA_input.groupby('Parameters').agg(['mean',f_ste]).T.loc['SI']
II1_df = SA_input.groupby('Parameters').agg(['mean',f_ste]).T.loc['II1']
II2_df = SA_input.groupby('Parameters').agg(['mean',f_ste]).T.loc['II2']
LI_df = SA_input.groupby('Parameters').agg(['mean',f_ste]).T.loc['LI']
SA_tot_df = SA_input.groupby('Parameters').agg(['mean',f_ste]).T.loc['SA_total']

SA_toPlot = pd.concat([SI_df, II1_df, II2_df, LI_df, SA_tot_df]).T
SA_toPlot.columns =␣
 ↪['SI_mean','SI_ste','II1_mean','II1_ste','II2_mean','II2_ste','LI_mean','LI_ste',
                    'SA_tot_mean','SA_tot_ste']
SA_toPlot = SA_toPlot.sort_values('SA_tot_mean',ascending=False)
labels = SA_toPlot.index
print('\nRanked parameters across species: '+str(labels.values.tolist()))
labels = ['SDD$_d$','LDD$_d$','FEC$_{max}$','GERM$_p$']

x = np.arange(len(labels)) # the label locations
width = 0.2  # the width of the bars

f, ax = plt.subplots()
f.set_figheight(3.54)
f.set_figwidth(3.54)
f.dpi = 300

rects1 = ax.bar(x - 2*width, SA_toPlot['SI_mean'], width,␣
 ↪yerr=SA_toPlot['SI_ste'], label='SI')
rects2 = ax.bar(x - width, SA_toPlot['II1_mean'], width,␣
 ↪yerr=SA_toPlot['II1_ste'], label='II1')
rects3 = ax.bar(x, SA_toPlot['II2_mean'], width, yerr=SA_toPlot['II2_ste'],␣
 ↪label='II2')
rects4 = ax.bar(x + width, SA_toPlot['LI_mean'], width,␣
 ↪yerr=SA_toPlot['LI_ste'], label='LI')

# Add some text for labels, title and custom x-axis tick labels, etc.
ax.set_ylabel('Sensitivity Analysis Index')
ax.set_xticks(x)
ax.set_xticklabels(labels)
ax.legend(frameon=False)
plt.xticks(rotation=45)

# Export figure
plt.savefig('Figure_1.png', bbox_inches='tight', dpi=300)
```

```
Parameter uncertainty mean:
                     UI
Parameters
```

```
FEC_max     3470.823529
GERM_p        27.294118
LDD_d        775.352941
SDD_d         61.521765
Parameter uncertainty standard deviation:
                     UI
Parameters
FEC_max     9339.379083
GERM_p        19.639771
LDD_d       1627.992166
SDD_d         34.656158

Ranked parameters across species: ['SDD_d', 'LDD_d', 'FEC_max', 'GERM_p']
```



8.1.1 Species-specific parameter ranking

[8]:
```
# Calculate a summary index as the summation of all others
SA_input['SA_total'] = SA_input['SI'] + SA_input['II1'] + SA_input['II2'] +␣
 ↪SA_input['LI']

species = SA_input.Species.unique()
for sp in species :
    sub_sp = SA_input[SA_input['Species']==sp]
    sub_sp = sub_sp.sort_values('SA_total',ascending=False)
    labels = sub_sp.Parameters
    print('Species: '+str(sp)+' --> Ranks 1: '+labels.values[0]+
          ' | 2: '+labels.values[1]+' | 2: '+labels.values[2]+
          ' | 3: '+labels.values[3])
```

```
Species: Abi_alb --> Ranks 1: SDD_d | 2: FEC_max | 2: LDD_d | 3: GERM_p
Species: Bet_pen --> Ranks 1: GERM_p | 2: SDD_d | 2: LDD_d | 3: FEC_max
Species: Bet_pub --> Ranks 1: GERM_p | 2: SDD_d | 2: LDD_d | 3: FEC_max
Species: Car_bet --> Ranks 1: SDD_d | 2: LDD_d | 2: GERM_p | 3: FEC_max
Species: Cor_ave --> Ranks 1: FEC_max | 2: SDD_d | 2: LDD_d | 3: GERM_p
Species: Fag_syl --> Ranks 1: SDD_d | 2: FEC_max | 2: LDD_d | 3: GERM_p
Species: Fra_exc --> Ranks 1: SDD_d | 2: LDD_d | 2: GERM_p | 3: FEC_max
Species: Pic_abi --> Ranks 1: LDD_d | 2: SDD_d | 2: GERM_p | 3: FEC_max
Species: Pic_sit --> Ranks 1: SDD_d | 2: LDD_d | 2: GERM_p | 3: FEC_max
Species: Pin_syl --> Ranks 1: FEC_max | 2: SDD_d | 2: LDD_d | 3: GERM_p
Species: Pin_hal --> Ranks 1: LDD_d | 2: FEC_max | 2: GERM_p | 3: SDD_d
Species: Que_coc --> Ranks 1: LDD_d | 2: SDD_d | 2: FEC_max | 3: GERM_p
Species: Que_ile --> Ranks 1: LDD_d | 2: FEC_max | 2: SDD_d | 3: GERM_p
Species: Que_pub --> Ranks 1: FEC_max | 2: SDD_d | 2: GERM_p | 3: LDD_d
Species: Que_rob --> Ranks 1: LDD_d | 2: FEC_max | 2: SDD_d | 3: GERM_p
Species: Til_cor --> Ranks 1: SDD_d | 2: LDD_d | 2: GERM_p | 3: FEC_max
Species: Ulm_gla --> Ranks 1: SDD_d | 2: LDD_d | 2: GERM_p | 3: FEC_max
```

8.1.2 Species-specific linear regression statistics

Additionally, we conducted species-specific linear regression analyses of the type $y \sim x_i$, such that every change of one parameter x_i unit translates to a change of migration rate (y) given by the slope coefficient of the regression (i.e. a slope value of 1 should correspond to $LI_i \sim 1$).

[9]:
```
# Calculate linear regression statistics
def lm_slope(group):
    x = group['value']
    y = group['MigRates']
    slope, intercept, r, p, se = stats.linregress(x, y)
    return slope

def lm_intercept(group):
    x = group['value']
    y = group['MigRates']
    slope, intercept, r, p, se = stats.linregress(x, y)
    return intercept

def lm_r(group):
    x = group['value']
    y = group['MigRates']
    slope, intercept, r, p, se = stats.linregress(x, y)
    return r

def lm_pvalue(group):
    x = group['value']
    y = group['MigRates']
    slope, intercept, r, p, se = stats.linregress(x, y)
    return p

# Format table to longer format (from 5-points)
long_df = SA_input[['Species','Parameters','Min','25th','Mean','75th','Max']]
long_df = long_df.melt(id_vars=['Species','Parameters'])
long_df.drop('variable', axis='columns', inplace=True)
mig_rates =␣
 ↪SA_input[['Species','Parameters','MigRate_Min','MigRate_25th','MigRate_Mean','MigRate_75th','
mig_rates = mig_rates.melt(id_vars=['Species','Parameters'])
long_df['MigRates'] = mig_rates['value']

# Calculate slopes
lm_stats_df = long_df.groupby(['Parameters','Species'])[['value','MigRates']].
 ↪apply(lm_slope)
lm_stats_df = lm_stats_df.reset_index()

intercepts = long_df.groupby(['Parameters','Species'])[['value','MigRates']].
 ↪apply(lm_intercept)
intercepts = intercepts.reset_index()

r_coeffs = long_df.groupby(['Parameters','Species'])[['value','MigRates']].
 ↪apply(lm_r)
r_coeffs = r_coeffs.reset_index()

pvalues = long_df.groupby(['Parameters','Species'])[['value','MigRates']].
 ↪apply(lm_pvalue)
pvalues = pvalues.reset_index()

lm_stats_df.columns = ['Parameters','Species','slope']
lm_stats_df['intercept'] = intercepts.iloc[:,2]
lm_stats_df['r2'] = r_coeffs.iloc[:,2]**2
lm_stats_df['pvalue'] = pvalues.iloc[:,2]
lm_stats_df.groupby('Parameters').mean()

# Print results
print('Linear regression summary:')
lm_stats_df.groupby('Parameters').describe()
```

```
Linear regression summary:
```

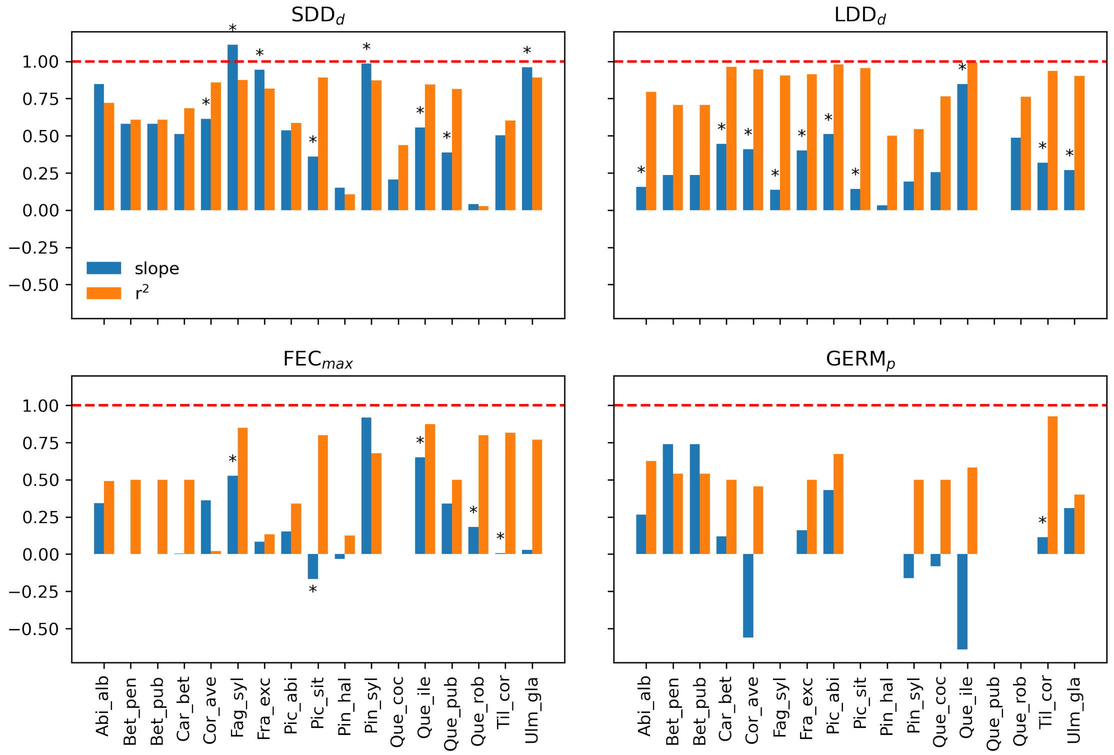
[9]:

```
           slope                                                              \
           count      mean       std       min       25%       50%       75%
Parameters
FEC_max     17.0  0.199839  0.287580 -0.168000  0.000085  0.083333  0.341935
GERM_p      17.0  0.084665  0.365090 -0.640000  0.000000  0.000000  0.266667
```

```
LDD_d       17.0  0.298626  0.206200  0.000000  0.156023  0.256000  0.409231
SDD_d       17.0  0.580878  0.306906  0.041365  0.388151  0.555311  0.848000

                  intercept              ...        r2            pvalue  \
                 max      count        mean  ...       75%       max   count
Parameters                                   ...
FEC_max     0.918919       17.0  39.734360  ...  0.798913  0.874379    17.0
GERM_p      0.740000       17.0  47.836903  ...  0.541535  0.925926    17.0
LDD_d       0.848000       17.0 -14.141041  ...  0.945202  0.991966    17.0
SDD_d       1.111644       17.0  -5.000152  ...  0.858112  0.892193    17.0

                                                                        \
                mean       std       min       25%       50%       75%
Parameters
FEC_max     0.260878  0.294861  0.019661  0.040866  0.181690  0.303056
GERM_p      0.402671  0.400843  0.008754  0.156318  0.181690  1.000000
LDD_d       0.099438  0.238192  0.000306  0.005536  0.013064  0.074067
SDD_d       0.143831  0.216615  0.015539  0.023730  0.068449  0.122752


                 max
Parameters
FEC_max     1.000000
GERM_p      1.000000
LDD_d       1.000000
SDD_d       0.789803

[4 rows x 32 columns]
```

[10]:

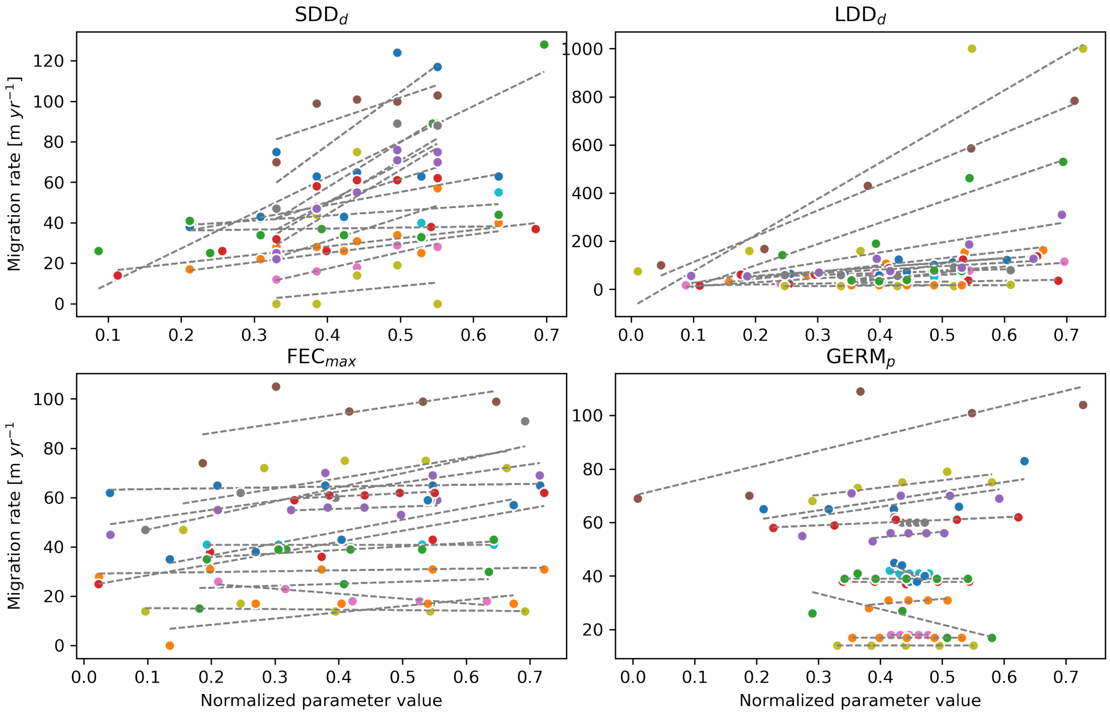
```
# Plot relationships migration rate vs. parameter values at the species level
f, axes = plt.subplots(2, 2, sharex=True, sharey=True)
(ax1, ax2), (ax3, ax4) = axes
f.set_figheight(7.25*1.5/1.6)
f.set_figwidth(7.25*1.5)
f.dpi = 300

headers = ['SDD_d','LDD_d','FEC_max','GERM_p']

for header,ax in zip(headers,axes.flatten()):
    lm_sub = lm_stats_df[lm_stats_df['Parameters']==header]

    labels = lm_sub.Species.unique()
    x = np.arange(len(labels))
    width = 0.375

    rects1 = ax.bar(x - width/2, lm_sub.slope, width, label='slope')
    rects2 = ax.bar(x + width/2, lm_sub.r2, width, label='r$^2$')
    # Add asterisks for significance
    sig_label = ['*' if (e < 0.05) else ('**' if (e < 0.01) else '') for e in␣
 ↪lm_sub.pvalue]
    ax.bar_label(rects1, labels=sig_label, padding=3)
    ax.axhline(y=1, color='r', linestyle='--')

    # Add headers
    if header == 'SDD_d' :
        ax.set_title('SDD$_d$')
        ax.legend(frameon=False)
    if header == 'LDD_d' :
        ax.set_title('LDD$_d$')
    if header == 'FEC_max' :
        ax.set_title('FEC$_{max}$')
        ax.set_xticks(x)
        ax.set_xticklabels(labels)
        ax.set_xticklabels(labels, rotation=90)
    if header == 'GERM_p' :
        ax.set_title('GERM$_p$')
        ax.set_xticks(x)
        ax.set_xticklabels(labels)
        ax.set_xticklabels(labels, rotation=90)
f.subplots_adjust(wspace=0.1)

plt.savefig('Figure_S2.png', bbox_inches='tight', dpi=300)
```



8.1.3 Species-specific relationship of migration rate vs. parameter values

[11]:

```
def scatter_plot_par(par_df, sub_plot):
    species = par_df['Species']
    for sp in species :
        # Normalize parameter values
        sp_sub = par_sub[par_sub['Species']==sp]
        norm_value = preprocessing.normalize([sp_sub['value']])
        x, y = norm_value[0], sp_sub['MigRates']
        sub_plot.plot(x, y, marker="o", linestyle="", markeredgecolor='white')
        slope, intercept = np.polyfit(x, y, 1)
        sub_plot.plot(x, slope*x+intercept, '--', color='grey', linewidth=1)

        # Add headers
        if header == 'SDD_d' :
            sub_plot.set_title('SDD$_d$')
            sub_plot.set_ylabel('Migration rate [m $yr^{-1}$]')
        if header == 'LDD_d' :
            sub_plot.set_title('LDD$_d$')
        if header == 'FEC_max' :
            sub_plot.set_title('FEC$_{max}$')
            sub_plot.set_xlabel('Normalized parameter value')
            sub_plot.set_ylabel('Migration rate [m $yr^{-1}$')
        if header == 'GERM_p' :
            sub_plot.set_title('GERM$_p$')
            sub_plot.set_xlabel('Normalized parameter value')

f, ax = plt.subplots()
f.set_figheight(3.54*2.)
f.set_figwidth(3.54*2.5)
f.set_figheight(7.25*1.5/1.6)
f.set_figwidth(7.25*1.5)
f.dpi = 300

# Place the plots in the plane
plot1 = plt.subplot2grid((2, 2), (0, 0))
plot2 = plt.subplot2grid((2, 2), (0, 1))
plot3 = plt.subplot2grid((2, 2), (1, 0))
plot4 = plt.subplot2grid((2, 2), (1, 1))

headers = ['SDD_d','LDD_d','FEC_max','GERM_p']

# SDD
par_sub = long_df[long_df['Parameters']==headers[0]]
header = headers[0]
scatter_plot_par(par_sub, plot1)

# LDD
par_sub = long_df[long_df['Parameters']==headers[1]]
header = headers[1]
scatter_plot_par(par_sub, plot2)

# FEC_max
par_sub = long_df[long_df['Parameters']==headers[2]]
header = headers[2]
scatter_plot_par(par_sub, plot3)

# GERM_p
par_sub = long_df[long_df['Parameters']==headers[3]]
header = headers[3]
scatter_plot_par(par_sub, plot4)

f.subplots_adjust(wspace=0.1)

plt.savefig('Figure_S3.png', bbox_inches='tight', dpi=300)
```



8.2 Extreme Value Analysis (EVA)

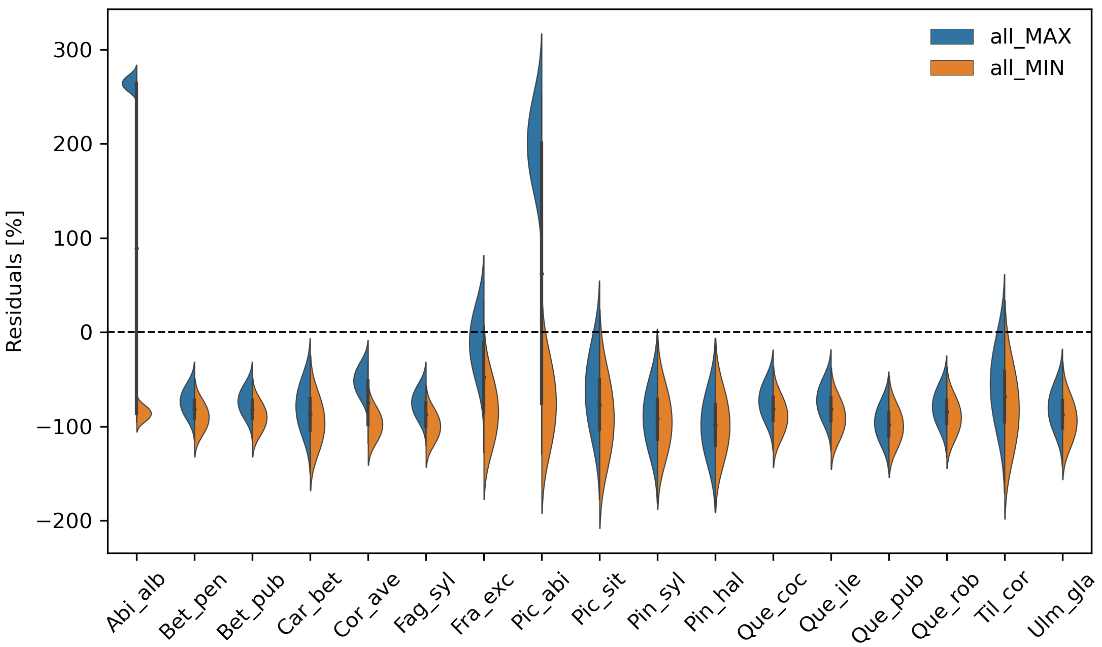
We fixed all influential parameters at their minimum (*all_MIN*) and maximum (*all_MAX*) values and calculated the corresponding errors with respect to observational data for each species. Species-specific performance was calculated with residuals (*res*) across the whole confidence range of observational values:

$$res = \frac{100}{\overline{obs}} \times (sim - obs)$$

where *obs* indicates all integer values from the lower to the upper boundary of migration rates estimates, and $\overline{obs}$ their average per species (Table 1).

[12]:
```
# Function to calculate overall RMSE and species-specific residuals of␣
 ↪observations vs. simulated migration values
def f_rmse_residuals(df_input) :

    # Calculate species-specific residuals
    Species = df_input['Species'].unique()
    Parameter_combo = df_input['Parameter_combo'].unique()
    if (Parameter_combo[2] == 'all_MAX_opt'):
        Parameter_combo = ['all_MAX','all_MIN']
    residuals_df = pd.DataFrame()
    for sp in Species :
        df_sp = df_input[df_input['Species']==sp]
        column_names = df_sp.columns.values.tolist()
        all_obs = np.arange(df_sp['OBS_MIN'].unique(),df_sp['OBS_MAX'].
↪unique(),1)
        mean_obs = all_obs.mean()
        for par in Parameter_combo :
            par_df = df_sp.loc[df_sp['Parameter_combo']==par]
            sim = par_df['SIM']
            check_na = pd.isna(sim)
            if (len(sim) != 1):
                sim = sim.reset_index()
                sim = sim['SIM'][0]
            if (check_na.any()) :
                sim = sim
            else :
                sim = int(sim)
            df = pd.DataFrame(index=np.arange(0,len(all_obs),1))
            if (column_names[2] == 'shape_par') :
                df.insert(0,'Species',sp)
                df.insert(1,'Parameter_combo',par)
                df.insert(2,'shape_par',float(par_df.shape_par.unique()))
                df.insert(3,'sim',sim)
                df.insert(4,'obs',all_obs)
            else :
                df.insert(0,'Species',sp)
                df.insert(1,'Parameter_combo',par)
                df.insert(2,'sim',sim)
                df.insert(3,'obs',all_obs)
            df['residuals'] = 100/mean_obs * (df['sim']-df['obs'])
            residuals_df = pd.concat([residuals_df,df])

    # Calculate average and standard deviation of residuals
    if (column_names[2] == 'shape_par') :
        df_output =␣
↪residuals_df[['Species','Parameter_combo','shape_par','residuals']].
↪groupby(['Species','Parameter_combo','shape_par']).agg(['mean','std']).
↪reset_index()
        df_output.columns =␣
↪['Species','Parameter_combo','shape_par','residuals_mean','residuals_std']
    else :
        df_output = residuals_df[['Species','Parameter_combo','residuals']].
↪groupby(['Species','Parameter_combo']).agg(['mean','std']).reset_index()
        df_output.columns =␣
↪['Species','Parameter_combo','residuals_mean','residuals_std']

    return [df_output, residuals_df]
```

```
# Function to make a violinplot of species-specific residuals
# input dataframe should be formatted with f_rmse_residuals()
def make_plot(residuals_df) :
    fig = plt.figure(figsize=(7.5,4.5))
    fig.dpi = 300
    fig = plt.axes()
    plt.xticks(rotation=45)
    if (len(residuals_df['Parameter_combo'].unique())==2) :
        fig = sns.violinplot(x='Species', y='residuals', hue='Parameter_combo',␣
 ↪data=residuals_df,
                             bw=1., width=0.5, linewidth=0.5, split=True,␣
 ↪dropna=True)
    else :
        standard_pal = sns.color_palette("tab10",5)
        standard_pal[4] = "#95a5a6"
        fig = sns.violinplot(x='Species', y='residuals', hue='Parameter_combo',␣
 ↪bw=1., palette=standard_pal,
                             data=residuals_df,  width=0.65, linewidth=0.,␣
 ↪dropna=True)
    fig.legend(frameon=False)
    fig.legend_.set_title(None)
    fig.set_xlabel(None)
    fig.set_ylabel('Residuals [%]')
    fig.axhline(0, color='black', ls='--', linewidth=0.9)
    fig_out = fig.get_figure()
    plt.tight_layout()
    return fig_out

# Make dataframe with RMSE and species-specific residuals (mean and standard␣
 ↪deviation)
df_output = f_rmse_residuals(EVA_input)[0]
residuals_df = f_rmse_residuals(EVA_input)[1]

# Export to CSV
df_output.to_csv('Table_S2.csv', sep=';', index=False)

# Make figure
fig_out = make_plot(residuals_df)

# Export figure
fig_out.savefig('Figure_2.png', dpi=300, format='png')
```



8.2.1 Overall error of all_MIN and all_MAX

Error across species was calculated as root-mean-square-error (RMSE) was used to quantify the model performance across species:

$$RMSE = \frac{100}{\overline{obs}} \times \sqrt{\frac{\sum_{i=1}^{n}(sim_i - obs_i)^2}{n}}$$

where n is the number of species, obs is the 75^{th} percentile value in the confidence range of observational values, and $\overline{obs}$ is the average across species.

```
[13]: allMIN_df = EVA_input[EVA_input['Parameter_combo']=='all_MIN']
allMIN_RMSE_perc = rmse_perc(allMIN_df.SIM,allMIN_df.OBS_75ile,allMIN_df.
 ↪OBS_50ile.mean())

allMAX_df = EVA_input[EVA_input['Parameter_combo']=='all_MAX']
allMAX_RMSE_perc = rmse_perc(allMAX_df.SIM,allMAX_df.OBS_75ile,allMAX_df.
 ↪OBS_50ile.mean())

print("Root Mean Square Error for all_MIN: "+str(round(allMIN_RMSE_perc,2)))
print("Root Mean Square Error for all_MAX: "+str(round(allMAX_RMSE_perc,2)))
```

```
Root Mean Square Error for all_MIN: 125.2
Root Mean Square Error for all_MAX: 111.01
```

8.2.2 Species-specific over/under/good estimations

Good estimations correspond to residuals within 10 m/yr from the average of observed migration speed (`residuals_mean`), over- and under-estimations correspond to positive and negative residuals outside of the 10 m/yr range, respectively.

```
[14]: good_est = df_output.loc[(df_output.residuals_mean < 10.) & (df_output.
       ↪residuals_mean > -10.)]
      bad_est = df_output.loc[(df_output.residuals_mean > 10.) | (df_output.
       ↪residuals_mean < -10.)]
      over_est = bad_est.loc[bad_est.residuals_mean > 0.]
      under_est = bad_est.loc[bad_est.residuals_mean < 0.]

      good_sp = good_est.Species.unique()
      for sp in good_sp :
          sp_sub = good_est[good_est['Species'] == sp]
          print('Good estimates: '+sp+' for '+str(sp_sub.Parameter_combo.values.
       ↪tolist()))

      over_sp = over_est.Species.unique()
      for sp in over_sp :
          sp_sub = over_est[over_est['Species'] == sp]
          print('Over-estimations: '+sp+' for '+str(sp_sub.Parameter_combo.values.
       ↪tolist()))

      under_sp = under_est.Species.unique()
      for sp in under_sp :
          sp_sub = under_est[under_est['Species'] == sp]
          print('Under-estimations: '+sp+' for '+str(sp_sub.Parameter_combo.values.
       ↪tolist()))

      # Calculate overall underestimation error (RMSE%)
      mean_MIN = under_est[under_est['Parameter_combo'] == 'all_MIN'].residuals_mean.
       ↪mean()
      std_MIN = under_est[under_est['Parameter_combo'] == 'all_MIN'].residuals_mean.
       ↪std()
      mean_MAX = under_est[under_est['Parameter_combo'] == 'all_MAX'].residuals_mean.
       ↪mean()
      std_MAX = under_est[under_est['Parameter_combo'] == 'all_MAX'].residuals_mean.
       ↪std()

      print('\n')
      print('Overall under-estimation errors: ')
      print('all_MIN | average error: '+str(round(mean_MIN,2))+' with S.D.:␣
       ↪'+str(round(std_MIN,2)))
      print('all_MAX | average error: '+str(round(mean_MAX,2))+' with S.D.:␣
       ↪'+str(round(std_MAX,2)))
```

```
Over-estimations: Abi_alb for ['all_MAX']
Over-estimations: Pic_abi for ['all_MAX']
Under-estimations: Abi_alb for ['all_MIN']
Under-estimations: Bet_pen for ['all_MAX', 'all_MIN']
Under-estimations: Bet_pub for ['all_MAX', 'all_MIN']
Under-estimations: Car_bet for ['all_MAX', 'all_MIN']
Under-estimations: Cor_ave for ['all_MAX', 'all_MIN']
Under-estimations: Fag_syl for ['all_MAX', 'all_MIN']
Under-estimations: Fra_exc for ['all_MAX', 'all_MIN']
Under-estimations: Pic_abi for ['all_MIN']
Under-estimations: Pic_sit for ['all_MAX', 'all_MIN']
Under-estimations: Pin_hal for ['all_MAX', 'all_MIN']
Under-estimations: Pin_syl for ['all_MAX', 'all_MIN']
Under-estimations: Que_coc for ['all_MAX', 'all_MIN']
Under-estimations: Que_ile for ['all_MAX', 'all_MIN']
Under-estimations: Que_pub for ['all_MAX', 'all_MIN']
Under-estimations: Que_rob for ['all_MAX', 'all_MIN']
Under-estimations: Til_cor for ['all_MAX', 'all_MIN']
Under-estimations: Ulm_gla for ['all_MAX', 'all_MIN']


Overall under-estimation errors:
all_MIN | average error: -91.5 with S.D.: 6.57
all_MAX | average error: -70.99 with S.D.: 21.19
```

8.2.3 Species-specific all_MIN vs. all_MAX

[15]:

```python
# Check significance for the difference all_MAX vs. all_MIN per species
# Mann-Whitney U Test
species = EVA_input.Species.unique().tolist()
best_EVA_df = pd.DataFrame()

# Store p-values for Bonferroni corrections
p_values = np.zeros(len(species))
pos = 0
for sp in species :
    best_EVA_sp = pd.DataFrame()
    best_EVA_sp['Species'] = sp
    sp_sub = residuals_df[residuals_df['Species']==sp]
    all_min = sp_sub[sp_sub['Parameter_combo']=='all_MIN']
    all_max = sp_sub[sp_sub['Parameter_combo']=='all_MAX']

    # Get p-values from Mann-Whitney U Test
    _, p = mannwhitneyu(all_min.residuals, all_max.residuals)
    p_values[pos] = p
    pos = pos+1
```

```
# Apply Bonferroni corrections for multiple comparison
p_adjusted = multipletests(p_values, method='bonferroni')[1]

# all_MIN vs. all_MAX based on corrected p-values
pos = 0
for sp in species :
    best_EVA_sp = pd.DataFrame()
    best_EVA_sp['Species'] = sp
    sp_sub = residuals_df[residuals_df['Species']==sp]
    all_min = sp_sub[sp_sub['Parameter_combo']=='all_MIN']
    all_max = sp_sub[sp_sub['Parameter_combo']=='all_MAX']
    max_opt = EVA_input.loc[(EVA_input['Parameter_combo']=='all_MAX_opt') &␣
 ↪(EVA_input['Species']==sp)]

    # Compare all_MIN and all_MAX
    all_max_res = abs(np.mean(all_max.residuals))
    all_min_res = abs(np.mean(all_min.residuals))
    if (all_max_res <= all_min_res) :
        best_sp = all_max
    else :
        best_sp = all_min

    # Check significance with low threshold
    p = p_adjusted[pos]
    pos = pos+1
    alpha = 1e-3
    if p > alpha:
        print(sp+ ' --> p-value: '+str(round(p,5))+' --> NOT-significant␣
 ↪difference')
    else:
        print(sp+ ' --> p-value: '+str(round(p,5))+' --> significant difference')

    # Select optimized set of parameters when possible
    if (len(max_opt) > 0):
        max_opt = max_opt.copy()
        max_opt['res'] = abs(max_opt.OBS_75ile - max_opt.SIM)
        best_sp = max_opt.loc[max_opt.res == min(max_opt.res)]

    # Select best parameter setting and corresponding simulated values
    best_EVA_sp['Parameter_combo'] = best_sp.Parameter_combo.unique()
    if (len(max_opt) > 0):
        best_EVA_sp['sim'] = best_sp.SIM.unique()
        best_EVA_sp['obs_75ile'] = best_sp.OBS_75ile.unique()
    else :
        best_EVA_sp['sim'] = best_sp.sim.unique()
        best_EVA_sp['obs_75ile'] = np.percentile(best_sp.obs, 75)
    best_EVA_df = pd.concat([best_EVA_df,best_EVA_sp])
```

```
best_EVA_df['Species'] = species
```

```
Abi_alb --> p-value: 0.0 --> significant difference
Bet_pen --> p-value: 0.0 --> significant difference
Bet_pub --> p-value: 0.0 --> significant difference
Car_bet --> p-value: 0.0 --> significant difference
Cor_ave --> p-value: 0.0 --> significant difference
Fag_syl --> p-value: 0.0 --> significant difference
Fra_exc --> p-value: 0.0 --> significant difference
Pic_abi --> p-value: 0.0 --> significant difference
Pic_sit --> p-value: 0.0 --> significant difference
Pin_syl --> p-value: 0.0 --> significant difference
Pin_hal --> p-value: 1.0 --> NOT-significant difference
Que_coc --> p-value: 0.0 --> significant difference
Que_ile --> p-value: 0.0 --> significant difference
Que_pub --> p-value: 0.08281 --> NOT-significant difference
Que_rob --> p-value: 0.0 --> significant difference
Til_cor --> p-value: 0.0 --> significant difference
Ulm_gla --> p-value: 0.0 --> significant difference
```

8.2.4 Best-performing EVA setting per species

[16]:
```
# Calculate RMSE%
rmse_mean = best_EVA_df.obs_75ile.mean()

# Add migration parameters for all_MAX settings
best_EVA_df['rmse'] = best_EVA_df.apply(lambda row: rmse_perc(row['sim'],␣
 ↪row['obs_75ile'], rmse_mean), axis = 1)
best_EVA_df['FEC_max'] = EVA_input.loc[EVA_input['Parameter_combo'] ==␣
 ↪'all_MAX','FEC_max'].to_list()
best_EVA_df['GERM_p'] = EVA_input.loc[EVA_input['Parameter_combo'] ==␣
 ↪'all_MAX','GERM_p'].to_list()
best_EVA_df['SDD_d'] = EVA_input.loc[EVA_input['Parameter_combo'] ==␣
 ↪'all_MAX','SDD_d'].to_list()
best_EVA_df['LDD_d'] = EVA_input.loc[EVA_input['Parameter_combo'] ==␣
 ↪'all_MAX','LDD_d'].to_list()

# Add migration parameters for all_MAX_opt species
sp_opt = best_EVA_df.loc[best_EVA_df['Parameter_combo'] ==␣
 ↪'all_MAX_opt','Species'].to_list()
for sp in sp_opt:
    best_sim = best_EVA_df.loc[best_EVA_df['Species'] == sp].sim
    best_pars = EVA_input.loc[(EVA_input['Parameter_combo'] == 'all_MAX_opt') &␣
 ↪(EVA_input['Species'] == sp)
                              & (EVA_input['SIM'] == int(best_sim))]
    best_EVA_df.loc[best_EVA_df['Species'] == sp,'FEC_max'] = int(best_pars.
 ↪FEC_max)
```

```
    best_EVA_df.loc[best_EVA_df['Species'] == sp,'GERM_p'] = int(best_pars.
↪GERM_p)
    best_EVA_df.loc[best_EVA_df['Species'] == sp,'SDD_d'] = int(best_pars.SDD_d)
    best_EVA_df.loc[best_EVA_df['Species'] == sp,'LDD_d'] = int(best_pars.LDD_d)
best_EVA_df
```

[16]:

```
   Species Parameter_combo  sim  obs_75ile        rmse  FEC_max  GERM_p  \
0  Abi_alb     all_MAX_opt  305     287.50    2.663086       81      60
0  Bet_pen         all_MAX  179     734.25   84.495916    30000      30
0  Bet_pub         all_MAX  179     734.25   84.495916    30000      30
0  Car_bet         all_MAX  162     874.25  108.387602      705      80
0  Cor_ave         all_MAX  607    1374.25  116.757301       12      60
0  Fag_syl         all_MAX   63     274.25   32.147253       62      80
0  Fra_exc         all_MAX  312     424.25   17.081795       50      65
0  Pic_abi     all_MAX_opt  352     412.50    9.206669      163      95
0  Pic_sit         all_MAX  125     411.75   43.636567       75      80
0  Pin_syl         all_MAX  116    1274.25  176.258252       43      95
0  Pin_hal         all_MAX   16    1274.25  191.475887       43      60
0  Que_coc         all_MAX  110     449.25   51.625825       10      75
0  Que_ile         all_MAX  111     449.25   51.473649       50      95
0  Que_pub         all_MAX   17     449.25   65.778225       50      90
0  Que_rob         all_MAX   85     449.25   55.430234       50      95
0  Til_cor         all_MAX  147     411.75   40.288687      720      55
0  Ulm_gla         all_MAX  153     886.75  111.659394      949      65

   SDD_d  LDD_d
0    100    710
0    200    475
0    200    475
0    100    425
0     25   1500
0     25    200
0    100    725
0    100    780
0    100    800
0    100    250
0    100    250
0     25    300
0     25    300
0     25    300
0     25    300
0    100    374
0    100    350
```

9 Evaluation of model uncertainty

9.1 Implementation of fat-tailed seed dispersal kernels

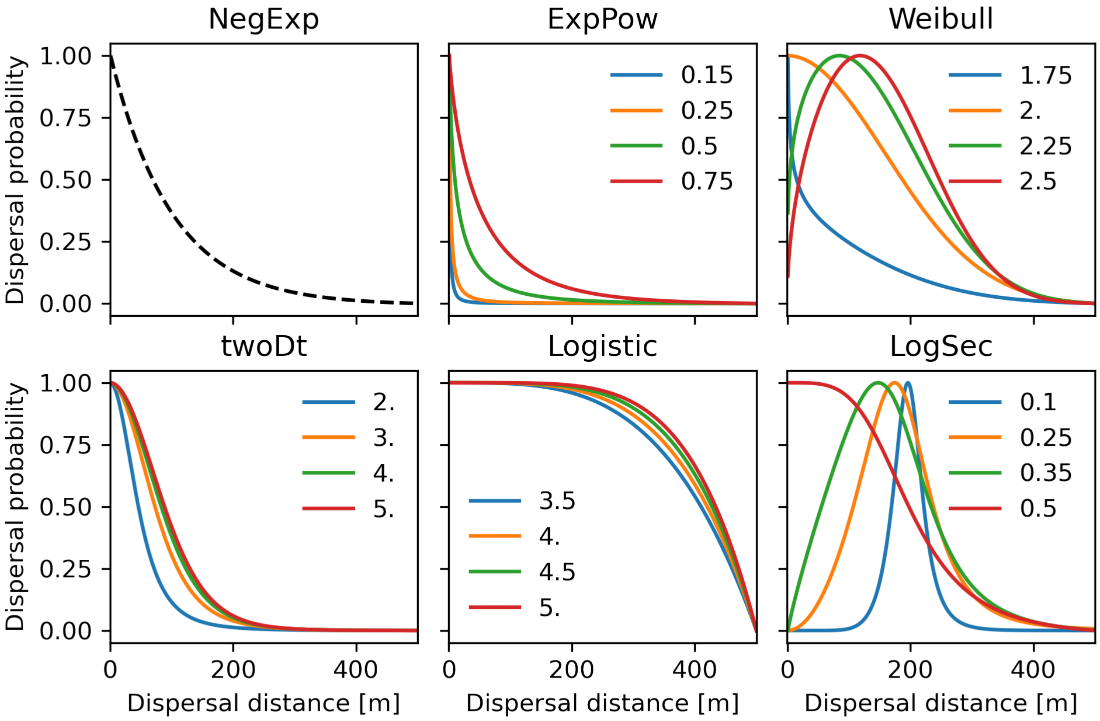
We implemented five additional fat-tailed kernels into the dispersal sub-model of LPJ-GM and run simulations with the best set of migration parameters found by EVA, while varying the shape parameter b in a meaningful range for each kernel.

Table 3. Probability density functions (pdf) for the default dispersal kernel in LPJ-GM (negative exponential) and five additional kernels. d = mean distance (in meters); a = scale parameter as a function of distance; b = shape parameter range to search for better representation of LDD events. Range boundaries are defined by the values for which pdf are mathematically significant and the corresponding tail is fat (Nathan et al., 2012). Table adapted from Bullock et al. (2017).

Kernel family	Probability density function	Scale parameter (a)	Shape parameter (b)	Weight of the tail
Negative exponential	$\frac{1}{2\pi a^2}\exp(-\frac{d}{a})$	$\frac{d}{2}$	-	Exponentially bounded
Exponential power	$\frac{b}{2\pi a^2\Gamma(\frac{2}{b})}\exp(-\frac{d^b}{a^b})$	$\frac{\Gamma(\frac{2}{b})}{\Gamma(\frac{3}{b})}$	$0-1$	Fat-tailed (for $b<1$) non-power law
Weibull	$\frac{b}{2\pi a^2}d^{b-2}\exp(-\frac{d^b}{a^b})$	$\frac{b}{\Gamma(\frac{1}{b})}d$	$0-2.5$	Fat-tailed non-power law
twoDt	$\frac{b-1}{\pi a^2}(1+\frac{d^2}{a^2}^{-b})$	$\frac{2}{\pi}\frac{\Gamma(b-1)}{\Gamma(b-\frac{3}{2})}d$	$1-5$	Fat-tailed power law
Logistic	$\frac{b}{2\pi a^2\Gamma(1-\frac{2}{b})}(1+\frac{d^b}{a^b})^{-1}$	$\frac{\Gamma(\frac{2}{b})\Gamma(1-\frac{2}{b})}{\Gamma(\frac{3}{b})\Gamma(1-\frac{3}{b})}d$	$2-5$	Fat-tailed power law
Log-hyperbolic secant	$\frac{1}{\pi^2 bd^2}(\frac{d}{a}^{\frac{1}{b}}+\frac{d}{a}^{-\frac{1}{b}})^{-1}$	d	$0-1$	Fat-tailed power law

[17]:

```
f, axes = plt.subplots(2, 3, sharex=True, sharey=True)
(ax1, ax2, ax3), (ax4, ax5, ax6) = axes
f.set_figheight(7.25/1.6)
f.set_figwidth(7.25)
f.dpi = 300

kernels = ['NegExp','ExpPow','Weibull','twoDt','Logistic','LogSec']
distance_range = np.linspace(1.,500.,500)
b = 1.
dd = 200.

for kernel,ax in zip(kernels,axes.flatten()):
    if kernel=='NegExp':
        def neg_exponential(dist, dd) :
            a = dd / 2
            if (a <= 0):
                return np.zeros(len(dist))
            dist_prob = 1 / (2 * np.pi * a**2) * np.exp(-dist / a)
            return scaler_0_1(dist_prob)
        x = neg_exponential(distance_range, dd)

    if kernel=='ExpPow':
        def exp_power(dist, dd, b) :
            a = (gamma(2/b) / gamma(3/b)) * dd
            if (a == 0. or b <= 0.):
                return np.zeros(len(dist))
            dist_prob = b / (2 * np.pi * a**2 * gamma(2/b)) * np.exp(-(dist**b /␣
↪a**b))
            return scaler_0_1(dist_prob)
        x1 = exp_power(distance_range, dd, 0.15)
        x2 = exp_power(distance_range, dd, 0.25)
        x3 = exp_power(distance_range, dd, 0.5)
        x4 = exp_power(distance_range, dd, 0.75)

    if kernel=='Weibull':
        def weibull(dist, dd, b) :
            a = (b / gamma(1/b)) * dd
            if (a <= 0. or b <= 0.):
                return np.zeros(len(dist))
            dist_prob = b / (2 * np.pi * a**2) * dist**(b - 2) * np.exp(-dist**b␣
↪/ a**b)
            return scaler_0_1(dist_prob)
        x1 = weibull(distance_range, dd, 1.75)
        x2 = weibull(distance_range, dd, 2.)
        x3 = weibull(distance_range, dd, 2.25)
        x4 = weibull(distance_range, dd, 2.5)

    if kernel=='twoDt':
        def twoDt(dist, dd, b) :
            a = (gamma(b - 1) / gamma(b - 3/2)) * 2/np.pi * dd
            if (a <= 0. or b <= 1.) :
                return np.zeros(len(dist))
            dist_prob = (b - 1) / (np.pi * a**2) * (1 + dist**2 / a**2)**(-b)
            return scaler_0_1(dist_prob)
        x1 = twoDt(distance_range, dd, 2.)
        x2 = twoDt(distance_range, dd, 3.)
        x3 = twoDt(distance_range, dd, 4.)
        x4 = twoDt(distance_range, dd, 5.)

    if kernel=='Logistic':
        def logistic(dist, dd, b) :
            a = (gamma(2/b) * gamma(1 - 2/b) / gamma(3/b) * gamma(1 - 3/b)) * dd
            if (a <= 0. or b <= 2.) :
                return np.zeros(len(dist))
            dist_prob = b / (2 * np.pi * a**2 * gamma(2/b) * gamma(1 - 2/b)) *␣
↪(1 + dist**b / a**b)**-1
            return scaler_0_1(dist_prob)
        x1 = logistic(distance_range, dd, 3.5)
        x2 = logistic(distance_range, dd, 4.)
        x3 = logistic(distance_range, dd, 4.5)
        x4 = logistic(distance_range, dd, 5.)

    if kernel=='LogSec':
        def log_sech(dist, dd, b) :
            a = dd
            if (a <= 0. or b <= 0.) :
                return np.zeros(len(dist))
            dist_prob = 1 / (np.pi**2 * b * dist**2) / ((dist / a)**(1 / b) +␣
 ↪(dist / a)**(-1 / b))
            return scaler_0_1(dist_prob)
        x1 = log_sech(distance_range, dd, 0.1)
        x2 = log_sech(distance_range, dd, 0.25)
        x3 = log_sech(distance_range, dd, 0.35)
        x4 = log_sech(distance_range, dd, 0.5)

    if kernel=='NegExp':
        ax.plot(distance_range,x,color='black',ls='--')
        ax.set_title(kernel)
        ax.set_xlim(-0.05,distance_range.max())
        ax.set_ylim(-0.05,1.05)
    else :
        ax.plot(distance_range,x1)
        ax.plot(distance_range,x2)
        ax.plot(distance_range,x3)
        ax.plot(distance_range,x4)
        if kernel=='ExpPow':
            ax.legend(['0.15','0.25','0.5','0.75'], frameon=False)
        if kernel=='Weibull':
            ax.legend(['1.75','2.','2.25','2.5'], frameon=False)
        if kernel=='twoDt':
            ax.legend(['2.','3.','4.','5.'], frameon=False)
        if kernel=='Logistic':
            ax.legend(['3.5','4.','4.5','5.'], frameon=False)
        if kernel=='LogSec':
            ax.legend(['0.1','0.25','0.35','0.5'], frameon=False)

    ax.grid(False)
    ax.set_xlim(-0.05,distance_range.max())
    ax.set_ylim(-0.05,1.05)
    ax.set_title(kernel)

ax4.set_xlabel('Dispersal distance [m]')
ax5.set_xlabel('Dispersal distance [m]')
ax6.set_xlabel('Dispersal distance [m]')
ax1.set_ylabel('Dispersal probability')
ax4.set_ylabel('Dispersal probability')
f.subplots_adjust(wspace=0.1)

plt.savefig('Figure_3.png', bbox_inches='tight', dpi=300)
```



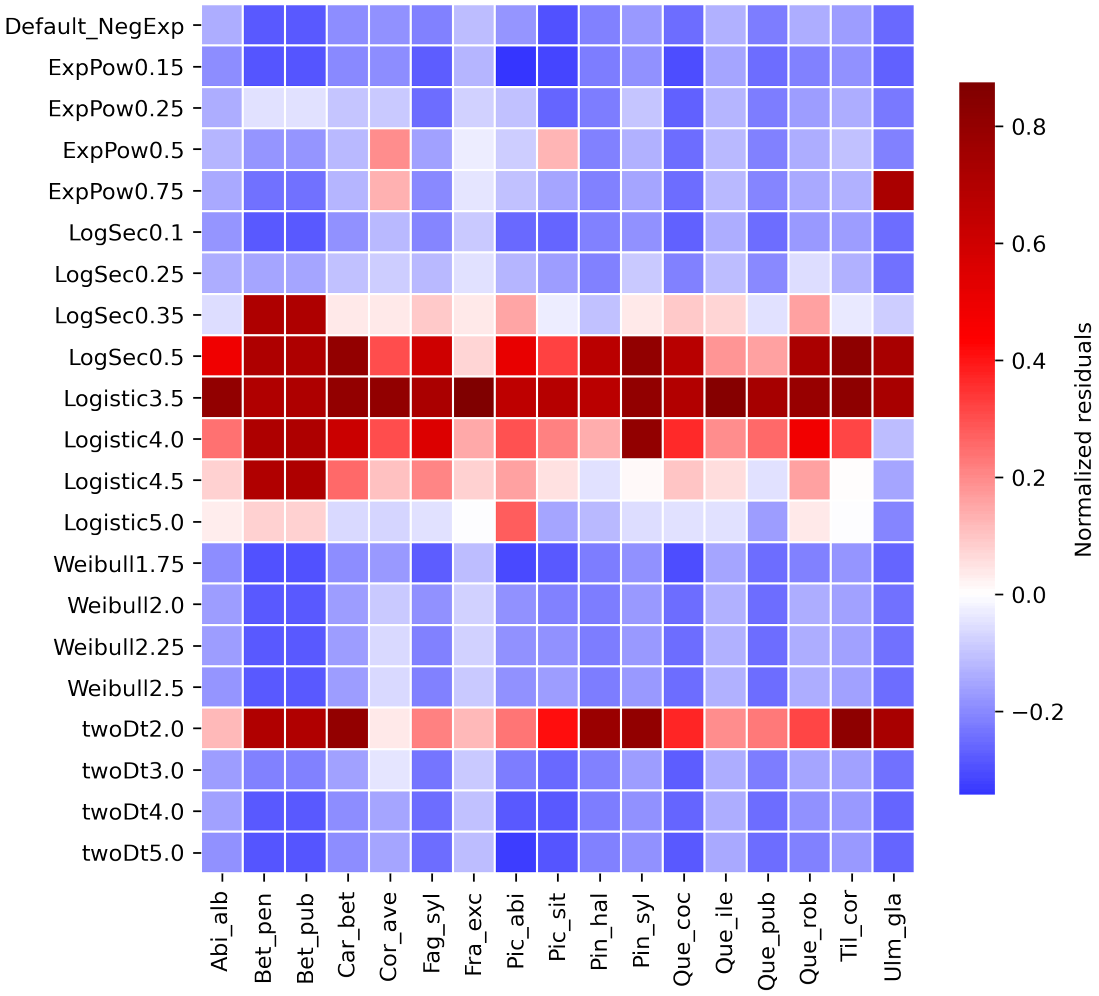
9.2 Species-specific performance for different shape parameters

We explored species- and kernel-specific parameter space of the shape parameter b by looking at the mean-normalized residuals of simulated migration rates with respect to the 75th percentiles of observed values.

[18]:

```python
# Calculate residuals to the 75th percentile observations
KA_input['residuals_75ile'] =  KA_input.SIM - KA_input.OBS_75ile
newkernels_df = KA_input.copy()

# Scale residuals at the species-level
species = newkernels_df.Species.unique()
res_mean_norm_df = pd.DataFrame()
for sp in species :
    sp_df = newkernels_df[newkernels_df['Species'] == sp]
    res_mean_norm = mean_norm(sp_df.residuals_75ile)
    res_mean_norm = pd.DataFrame(res_mean_norm)
    res_mean_norm_df = pd.concat([res_mean_norm_df,res_mean_norm])
newkernels_df = newkernels_df.copy()
newkernels_df['residuals_scaled'] = res_mean_norm_df.values

# Merge labels of kernel function and shape parameter
newkernels_df['shape_par'] = list(map(str, newkernels_df['shape_par']))
```

```
newkernels_df['kernel_combo'] = newkernels_df.kernel_fun.str.cat(newkernels_df.
 ↪shape_par)
newkernels_df.loc[newkernels_df.kernel_fun == 'NegExp','kernel_combo'] =␣
 ↪'Default_NegExp'

# Select representetive shape parameter values to plot
ExpPow_df = newkernels_df.loc[(newkernels_df.kernel_fun == 'ExpPow') &
                  (newkernels_df['shape_par'].isin(['0.15','0.25','0.5','0.
 ↪75']))]
Weibull_df = newkernels_df.loc[(newkernels_df.kernel_fun == 'Weibull') &
                  (newkernels_df['shape_par'].isin(['1.75','2.0','2.25','2.5']))]
twoDt_df = newkernels_df.loc[(newkernels_df.kernel_fun == 'twoDt') &
                  (newkernels_df['shape_par'].isin(['2.0','3.0','4.0','5.0']))]
Logistic_df = newkernels_df.loc[(newkernels_df.kernel_fun == 'Logistic') &
                  (newkernels_df['shape_par'].isin(['3.5','4.0','4.5','5.0']))]
LogSec_df = newkernels_df.loc[(newkernels_df.kernel_fun == 'LogSec') &
                  (newkernels_df['shape_par'].isin(['0.1','0.25','0.35','0.5']))]
NegExp_df = newkernels_df[newkernels_df.kernel_fun == 'NegExp']
kernel_sel_df = pd.
 ↪concat([ExpPow_df,Weibull_df,twoDt_df,Logistic_df,LogSec_df,NegExp_df])

# Make dataframe to plot
heatmap_df = pd.DataFrame()
heatmap_df['Species'] = kernel_sel_df.Species.values
heatmap_df['Kernel'] = kernel_sel_df.kernel_combo.values
heatmap_df['Residuals'] = kernel_sel_df.residuals_scaled.values
heatmap_df.Residuals.astype(float)
heatmap_df = heatmap_df.pivot('Kernel', 'Species', 'Residuals')

# Plot heatmap
f, ax = plt.subplots()
f.set_figheight(3.54*2)
f.set_figwidth(3.54*2)
f.dpi = 300
heatmap = sns.heatmap(heatmap_df, cmap='seismic', center=0.00, linewidths=.5,
                      cbar_kws={"shrink": .82,'label': 'Normalized residuals'})
ax.set_ylabel('')
ax.set_xlabel('')

# Export figure
plt.savefig('Figure_4.png', bbox_inches='tight', dpi=300)
```



9.3 Best-fitted shape parameters

We selected the species-specific shape parameter values that significantly minimized residuals for each kernel function.

[19]:
```
# Calculate absolute residuals
newkernels_df = newkernels_df[newkernels_df['kernel_fun'] != 'NegExp']
newkernels_df = newkernels_df.copy()
newkernels_df['residuals_abs'] = abs(newkernels_df.residuals_75ile)

# Find the minimum residual per species per kernel
species = newkernels_df.Species.unique()
bestfit_newkernels_df = newkernels_df.groupby(['Species','kernel_fun']).
 ↪apply(min_residual)

# Format table for further analysis
```

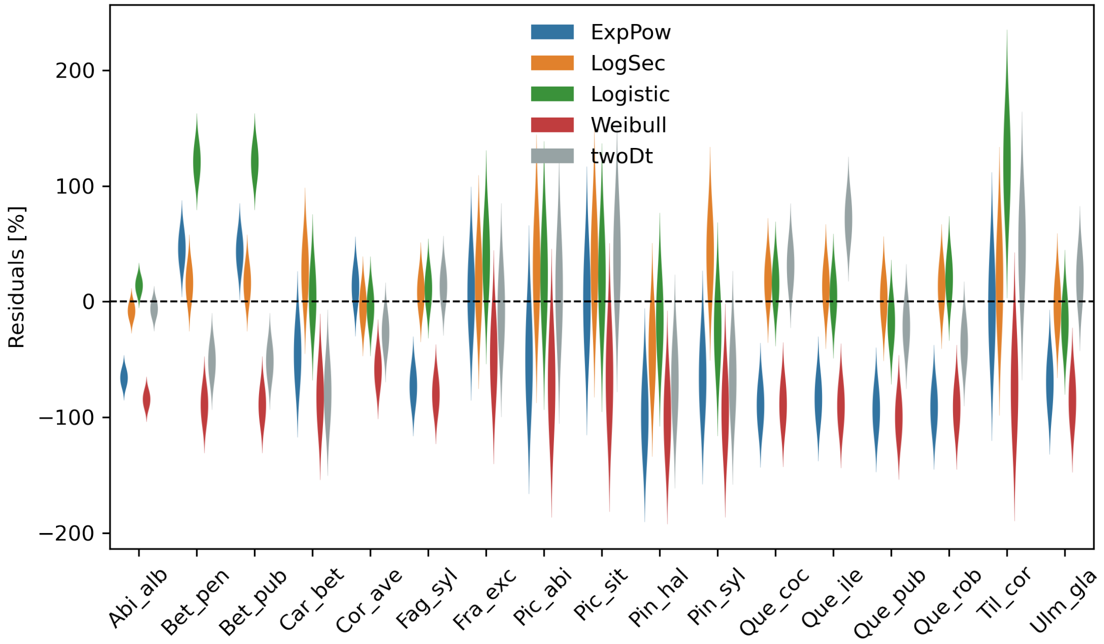
```
bestfit_newkernels_df.drop('Species', axis='columns', inplace=True)
bestfit_newkernels_df.drop('kernel_fun', axis='columns', inplace=True)
bestfit_newkernels_df = bestfit_newkernels_df.reset_index()
bestfit_newkernels_df.drop('level_2', axis='columns', inplace=True)
bestfit_newkernels_df.drop('residuals_scaled', axis='columns', inplace=True)
bestfit_newkernels_df.drop('kernel_combo', axis='columns', inplace=True)
bestfit_newkernels_df.drop('residuals_abs', axis='columns', inplace=True)
bestfit_newkernels_df = bestfit_newkernels_df.rename(columns={'kernel_fun':␣
 ↪'Parameter_combo'})

# Save table
bestfit_newkernels_df.to_csv('Table_S3.csv', sep=';', index=False)
```

9.4 Performance evaluation of new dispersal kernels

We evaluated the performance of the new kernels by calculating the error between simulated and observed migration rates for each species (residuals) and across species (RMSE). Additionally, we conducted a one-way ANOVA with post-hoc Tukey's HSD test among errors generated by newly-added kernels to verify whether one or more fat-tailed kernels improved the predictions across all species (RMSEs) or at the species level (residuals).

[20]:

```
# Make dataframe with RMSE and average and standard deviation of␣
 ↪species-specific residuals
df_output = f_rmse_residuals(bestfit_newkernels_df)[0]
residuals_df = f_rmse_residuals(bestfit_newkernels_df)[1]

# Export to CSV
df_output.to_csv('Table_S4.csv', sep=';', index=False)

## Species-specific residuals for the best-fitted shape parameter of each␣
 ↪fat-tailed kernel
# Make figure
fig_out = make_plot(residuals_df)

# Export figure
fig_out.savefig('Figure_5.png', dpi=300, format='png')
```



9.5 Overall error per kernel function

Error across species was calculated as root-mean-square-error (RMSE) was used to quantify the model performance across species:

$$RMSE = \frac{100}{\overline{obs}} \times \sqrt{\frac{\sum_{i=1}^{n}(sim_i - obs_i)^2}{n}}$$

where n is the number of species, obs is the 75^{th} percentile value in the confidence range of observational values, and $\overline{obs}$ is the average across species.

```
[21]: RMSE = rmse_perc(best_EVA_df.sim,best_EVA_df.obs_75ile,best_EVA_df.obs_75ile.
       ↪mean())
      print('NegExp RMSE: '+str(round(RMSE,2)))

      kernels = bestfit_newkernels_df.Parameter_combo.unique()
      for kernel in kernels:
          kernel_df =␣
       ↪bestfit_newkernels_df[bestfit_newkernels_df['Parameter_combo']==kernel]
          RMSE = rmse_perc(kernel_df.SIM,kernel_df.OBS_75ile,kernel_df.OBS_75ile.
       ↪mean())
          print(kernel+' RMSE: '+str(round(RMSE,2)))
```

```
NegExp RMSE: 89.94
ExpPow RMSE: 73.67
LogSec RMSE: 27.26
```

```
Logistic RMSE: 48.07
Weibull RMSE: 97.72
twoDt RMSE: 63.19
```

9.6 Select best-fitted kernel per species

We selected the kernel function that significantly minimized residuals for each species.

[22]:

```
species = residuals_df.Species.unique()
df_output['residuals_abs'] = df_output.residuals_mean.abs()

rmse_all_df = pd.DataFrame()
for sp in species :
    sp_res = df_output[df_output['Species'] == sp]
    sp_df = residuals_df[residuals_df['Species'] == sp]

    # RMSE% of each kernel
    rmse_df = pd.DataFrame()
    rmse_df['Species'] = sp
    kernels = sp_res.Parameter_combo
    rmse_df['kernel'] = kernels
    rmse_df['shape_par'] = sp_res.shape_par
    rmse_df['rmse'] = 0.
    rmse_df['Species'] = sp
    for kernel in kernels:
        kernel_df = sp_df[sp_df['Parameter_combo'] == kernel]
        RMSE = rmse_perc(kernel_df.sim,kernel_df.obs,kernel_df.obs.mean())
        rmse_df.loc[rmse_df['kernel'] == kernel, 'rmse'] = RMSE

    print(sp+' - Kernel ranking [RMSE%]')

    # Rank kernel performance based on absolute residual value
    kernels_rank = sp_res.sort_values('residuals_abs',ascending=True).
↪reset_index(drop=True).Parameter_combo
    print('1: '+str(kernels_rank[0])+' '+str(rmse_df.
↪loc[rmse_df['kernel']==kernels_rank[0]].rmse.values.round(2))+
          ' | 2: '+str(kernels_rank[1])+' '+str(rmse_df.
↪loc[rmse_df['kernel']==kernels_rank[1]].rmse.values.round(2))+
          ' | 3: '+str(kernels_rank[2])+' '+str(rmse_df.
↪loc[rmse_df['kernel']==kernels_rank[2]].rmse.values.round(2))+
          ' | 4: '+str(kernels_rank[3])+' '+str(rmse_df.
↪loc[rmse_df['kernel']==kernels_rank[3]].rmse.values.round(2))+
          ' | 5: '+str(kernels_rank[4])+' '+str(rmse_df.
↪loc[rmse_df['kernel']==kernels_rank[4]].rmse.values.round(2)))

    # Select best kernel
    best_kernel_sp = rmse_df.loc[rmse_df['kernel']==kernels_rank[0]]
    sim_value = sp_df[sp_df['Parameter_combo']==kernels_rank[0]].sim.unique()
    obs_values = sp_df[sp_df['Parameter_combo']==kernels_rank[0]].obs
    obs_75ile = np.percentile(obs_values, 75)
    best_kernel_sp = best_kernel_sp.copy()
    best_kernel_sp['sim'] = float(sim_value)
    best_kernel_sp['obs_75ile'] = float(obs_75ile)
    rmse_all_df = pd.concat([rmse_all_df,best_kernel_sp])
rmse_all_df = rmse_all_df.reset_index()
rmse_all_df.drop('index', axis='columns', inplace=True)
```

```
Abi_alb - Kernel ranking [RMSE%]
1: twoDt [7.99] | 2: LogSec [9.43] | 3: Logistic [14.98] | 4: ExpPow [65.97] |
5: Weibull [84.5]
Bet_pen - Kernel ranking [RMSE%]
1: LogSec [19.71] | 2: ExpPow [47.28] | 3: twoDt [52.81] | 4: Weibull [89.8] |
5: Logistic [121.58]
Bet_pub - Kernel ranking [RMSE%]
1: LogSec [19.71] | 2: ExpPow [44.82] | 3: twoDt [52.81] | 4: Weibull [89.8] |
5: Logistic [121.58]
Car_bet - Kernel ranking [RMSE%]
1: Logistic [19.63] | 2: LogSec [32.96] | 3: ExpPow [49.34] | 4: twoDt [80.98] |
5: Weibull [84.48]
Cor_ave - Kernel ranking [RMSE%]
1: LogSec [12.21] | 2: Logistic [12.21] | 3: ExpPow [17.51] | 4: twoDt [28.94] |
5: Weibull [59.52]
Fag_syl - Kernel ranking [RMSE%]
1: LogSec [14.19] | 2: Logistic [16.26] | 3: twoDt [18.03] | 4: ExpPow [74.06] |
5: Weibull [80.79]
Fra_exc - Kernel ranking [RMSE%]
1: ExpPow [25.75] | 2: twoDt [25.83] | 3: LogSec [30.06] | 4: Logistic [45.77] |
5: Weibull [53.95]
Pic_abi - Kernel ranking [RMSE%]
1: twoDt [32.9] | 2: Logistic [38.5] | 3: LogSec [42.21] | 4: ExpPow [58.97] |
5: Weibull [76.99]
Pic_sit - Kernel ranking [RMSE%]
1: ExpPow [31.15] | 2: Logistic [37.45] | 3: LogSec [45.69] | 4: twoDt [49.17] |
5: Weibull [72.51]
Pin_hal - Kernel ranking [RMSE%]
1: Logistic [29.2] | 2: LogSec [48.4] | 3: twoDt [73.25] | 4: ExpPow [101.17] |
5: Weibull [103.02]
Pin_syl - Kernel ranking [RMSE%]
1: Logistic [34.32] | 2: LogSec [48.32] | 3: ExpPow [70.03] | 4: twoDt [70.3] |
5: Weibull [97.29]
Que_coc - Kernel ranking [RMSE%]
1: Logistic [21.11] | 2: LogSec [23.4] | 3: twoDt [34.35] | 4: Weibull [90.15] |
5: ExpPow [90.65]
Que_ile - Kernel ranking [RMSE%]
1: Logistic [15.25] | 2: LogSec [19.53] | 3: twoDt [72.91] | 4: ExpPow [85.21] |
5: Weibull [91.14]
Que_pub - Kernel ranking [RMSE%]
1: LogSec [14.69] | 2: Logistic [22.81] | 3: twoDt [25.82] | 4: ExpPow [94.6] |
5: Weibull [101.04]
Que_rob - Kernel ranking [RMSE%]
1: LogSec [19.53] | 2: Logistic [24.8] | 3: twoDt [39.18] | 4: ExpPow [92.38] |
5: Weibull [92.38]
Til_cor - Kernel ranking [RMSE%]
1: ExpPow [31.41] | 2: LogSec [35.82] | 3: twoDt [57.41] | 4: Weibull [79.82] |
5: Logistic [123.11]
Ulm_gla - Kernel ranking [RMSE%]
1: LogSec [17.12] | 2: Logistic [24.42] | 3: twoDt [26.16] | 4: ExpPow [71.52] |
5: Weibull [86.66]
```

[23]:

```
good_est = df_output.loc[(df_output.residuals_mean < 10.) & (df_output.
 ↪residuals_mean > -10.)]
bad_est = df_output.loc[(df_output.residuals_mean > 10.) | (df_output.
 ↪residuals_mean < -10.)]
over_est = bad_est.loc[bad_est.residuals_mean > 0.]
under_est = bad_est.loc[bad_est.residuals_mean < 0.]

good_sp = good_est.Species.unique()
for sp in good_sp :
    sp_sub = good_est[good_est['Species'] == sp]
    print('Good estimates: '+sp+' for '+str(sp_sub.Parameter_combo.values.
 ↪tolist()))

over_sp = over_est.Species.unique()
for sp in over_sp :
    sp_sub = over_est[over_est['Species'] == sp]
    print('Over-estimations: '+sp+' for '+str(sp_sub.Parameter_combo.values.
 ↪tolist()))

under_sp = under_est.Species.unique()
for sp in under_sp :
    sp_sub = under_est[under_est['Species'] == sp]
    print('Under-estimations: '+sp+' for '+str(sp_sub.Parameter_combo.values.
 ↪tolist()))

# Calculate overall underestimation error (RMSE%)
mean_MIN = under_est[under_est['Parameter_combo'] == 'all_MIN'].residuals_mean.
 ↪mean()
std_MIN = under_est[under_est['Parameter_combo'] == 'all_MIN'].residuals_mean.
 ↪std()
mean_MAX = under_est[under_est['Parameter_combo'] == 'all_MAX'].residuals_mean.
 ↪mean()
```

```
std_MAX = under_est[under_est['Parameter_combo'] == 'all_MAX'].residuals_mean.
 ↪std()
```

```
Good estimates: Abi_alb for ['LogSec', 'twoDt']
Good estimates: Car_bet for ['Logistic']
Good estimates: Cor_ave for ['LogSec', 'Logistic']
Good estimates: Fag_syl for ['LogSec']
Good estimates: Fra_exc for ['ExpPow', 'twoDt']
Good estimates: Pic_sit for ['ExpPow']
Good estimates: Que_ile for ['Logistic']
Good estimates: Que_pub for ['LogSec']
Good estimates: Til_cor for ['ExpPow']
Good estimates: Ulm_gla for ['LogSec']
Over-estimations: Abi_alb for ['Logistic']
Over-estimations: Bet_pen for ['ExpPow', 'LogSec', 'Logistic']
Over-estimations: Bet_pub for ['ExpPow', 'LogSec', 'Logistic']
Over-estimations: Car_bet for ['LogSec']
Over-estimations: Cor_ave for ['ExpPow']
Over-estimations: Fag_syl for ['Logistic', 'twoDt']
Over-estimations: Fra_exc for ['LogSec', 'Logistic']
Over-estimations: Pic_abi for ['LogSec', 'Logistic', 'twoDt']
Over-estimations: Pic_sit for ['LogSec', 'Logistic', 'twoDt']
Over-estimations: Pin_syl for ['LogSec']
Over-estimations: Que_coc for ['LogSec', 'Logistic', 'twoDt']
Over-estimations: Que_ile for ['LogSec', 'twoDt']
Over-estimations: Que_rob for ['LogSec', 'Logistic']
Over-estimations: Til_cor for ['LogSec', 'Logistic', 'twoDt']
Over-estimations: Ulm_gla for ['twoDt']
Under-estimations: Abi_alb for ['ExpPow', 'Weibull']
Under-estimations: Bet_pen for ['Weibull', 'twoDt']
Under-estimations: Bet_pub for ['Weibull', 'twoDt']
Under-estimations: Car_bet for ['ExpPow', 'Weibull', 'twoDt']
Under-estimations: Cor_ave for ['Weibull', 'twoDt']
Under-estimations: Fag_syl for ['ExpPow', 'Weibull']
Under-estimations: Fra_exc for ['Weibull']
Under-estimations: Pic_abi for ['ExpPow', 'Weibull']
Under-estimations: Pic_sit for ['Weibull']
Under-estimations: Pin_hal for ['ExpPow', 'LogSec', 'Logistic', 'Weibull',
'twoDt']
Under-estimations: Pin_syl for ['ExpPow', 'Logistic', 'Weibull', 'twoDt']
Under-estimations: Que_coc for ['ExpPow', 'Weibull']
Under-estimations: Que_ile for ['ExpPow', 'Weibull']
Under-estimations: Que_pub for ['ExpPow', 'Logistic', 'Weibull', 'twoDt']
Under-estimations: Que_rob for ['ExpPow', 'Weibull', 'twoDt']
Under-estimations: Til_cor for ['Weibull']
Under-estimations: Ulm_gla for ['ExpPow', 'Logistic', 'Weibull']
```

9.7 Test species-specific kernel performance

We conducted a one-way ANOVA with post-hoc Tukey's HSD test among residuals generated by the newly-added kernels to verify whether one or more fat-tailed kernels improved the predictions at the species level.

[24]:

```python
for sp in species :
    sp_df = residuals_df[residuals_df['Species'] == sp]

    # Perform one-way ANOVA
    twoDt_res = sp_df[sp_df['Parameter_combo'] == '2Dt'].residuals
    exppow_res = sp_df[sp_df['Parameter_combo'] == 'ExpPow'].residuals
    logsec_res = sp_df[sp_df['Parameter_combo'] == 'LogSec'].residuals
    weibull_res = sp_df[sp_df['Parameter_combo'] == 'Weibull'].residuals
    logistic_res = sp_df[sp_df['Parameter_combo'] == 'Logistic'].residuals
    stats.f_oneway(twoDt_res,exppow_res,logsec_res,logistic_res,weibull_res)

    # Perform Tukey's test
    tukey = pairwise_tukeyhsd(endog=sp_df.residuals,
                              groups=sp_df.Parameter_combo,
                              alpha=0.05)
    print('=== '+sp+' ===')
    print(tukey)
```

```
=== Abi_alb ===
    Multiple Comparison of Means - Tukey HSD, FWER=0.05
===========================================================
 group1   group2  meandiff p-adj   lower    upper   reject
-----------------------------------------------------------
  ExpPow   LogSec  57.9235  0.001   55.0046  60.8424   True
  ExpPow Logistic  79.7814  0.001   76.8625  82.7004   True
  ExpPow  Weibull -18.5792  0.001  -21.4982 -15.6603   True
  ExpPow    twoDt   59.745  0.001   56.8261  62.6639   True
  LogSec Logistic  21.8579  0.001    18.939  24.7769   True
  LogSec  Weibull -76.5027  0.001  -79.4217 -73.5838   True
  LogSec    twoDt   1.8215 0.4283   -1.0974   4.7404  False
Logistic  Weibull -98.3607  0.001 -101.2796 -95.4417   True
Logistic    twoDt -20.0364  0.001  -22.9554 -17.1175   True
 Weibull    twoDt  78.3242  0.001   75.4053  81.2432   True
-----------------------------------------------------------
=== Bet_pen ===
    Multiple Comparison of Means - Tukey HSD, FWER=0.05
============================================================
 group1   group2   meandiff p-adj   lower     upper   reject
------------------------------------------------------------
  ExpPow   LogSec  -29.7237 0.001  -32.4147  -27.0327   True
  ExpPow Logistic   75.1307 0.001   72.4397   77.8217   True
  ExpPow  Weibull -135.0261 0.001 -137.7171 -132.3351   True
  ExpPow    twoDt  -97.5355 0.001 -100.2265  -94.8445   True
  LogSec Logistic  104.8544 0.001  102.1634  107.5454   True
  LogSec  Weibull -105.3025 0.001 -107.9935 -102.6115   True
  LogSec    twoDt  -67.8118 0.001  -70.5028  -65.1208   True
Logistic  Weibull -210.1568 0.001 -212.8478 -207.4658   True
Logistic    twoDt -172.6662 0.001 -175.3572 -169.9752   True
 Weibull    twoDt   37.4907 0.001   34.7997   40.1817   True
------------------------------------------------------------
=== Bet_pub ===
    Multiple Comparison of Means - Tukey HSD, FWER=0.05
============================================================
 group1   group2   meandiff p-adj   lower     upper   reject
------------------------------------------------------------
  ExpPow   LogSec  -27.1845 0.001  -29.8755  -24.4935   True
  ExpPow Logistic   77.6699 0.001   74.9789   80.3609   True
  ExpPow  Weibull -132.4869 0.001 -135.1779 -129.7959   True
  ExpPow    twoDt  -94.9963 0.001  -97.6873  -92.3053   True
  LogSec Logistic  104.8544 0.001  102.1634  107.5454   True
  LogSec  Weibull -105.3025 0.001 -107.9935 -102.6115   True
  LogSec    twoDt  -67.8118 0.001  -70.5028  -65.1208   True
Logistic  Weibull -210.1568 0.001 -212.8478 -207.4658   True
Logistic    twoDt -172.6662 0.001 -175.3572 -169.9752   True
 Weibull    twoDt   37.4907 0.001   34.7997   40.1817   True
------------------------------------------------------------
=== Car_bet ===
     Multiple Comparison of Means - Tukey HSD, FWER=0.05
=============================================================
 group1   group2   meandiff p-adj    lower     upper   reject
-------------------------------------------------------------
  ExpPow   LogSec   72.1815  0.001   68.8534   75.5095   True
  ExpPow Logistic   49.2328  0.001   45.9047   52.5609   True
  ExpPow  Weibull  -36.8245  0.001  -40.1526  -33.4965   True
  ExpPow    twoDt  -33.2221  0.001  -36.5502  -29.8941   True
  LogSec Logistic  -22.9486  0.001  -26.2767  -19.6205   True
  LogSec  Weibull  -109.006  0.001 -112.3341 -105.6779   True
  LogSec    twoDt -105.4036  0.001 -108.7317 -102.0755   True
Logistic  Weibull  -86.0574  0.001  -89.3855  -82.7293   True
Logistic    twoDt   -82.455  0.001  -85.7831  -79.1269   True
 Weibull    twoDt    3.6024 0.0262    0.2743    6.9305   True
-------------------------------------------------------------
=== Cor_ave ===
   Multiple Comparison of Means - Tukey HSD, FWER=0.05
=========================================================
 group1   group2  meandiff p-adj  lower    upper   reject
---------------------------------------------------------
  ExpPow   LogSec -17.1269 0.001 -19.1232 -15.1305   True
  ExpPow Logistic -17.1269 0.001 -19.1232 -15.1305   True
  ExpPow  Weibull -71.5486 0.001 -73.5449 -69.5523   True
  ExpPow    twoDt -39.6959 0.001 -41.6922 -37.6996   True
  LogSec Logistic      0.0   0.9  -1.9963   1.9963  False
  LogSec  Weibull -54.4218 0.001 -56.4181 -52.4254   True
  LogSec    twoDt  -22.569 0.001 -24.5654 -20.5727   True
Logistic  Weibull -54.4218 0.001 -56.4181 -52.4254   True
Logistic    twoDt  -22.569 0.001 -24.5654 -20.5727   True
 Weibull    twoDt  31.8527 0.001  29.8564  33.8491   True
---------------------------------------------------------
=== Fag_syl ===
   Multiple Comparison of Means - Tukey HSD, FWER=0.05
==========================================================
 group1   group2  meandiff p-adj   lower    upper   reject
----------------------------------------------------------
  ExpPow   LogSec  81.3627  0.001  76.8605   85.865   True
  ExpPow Logistic  84.5691  0.001  80.0669  89.0714   True
  ExpPow  Weibull  -6.8136  0.001 -11.3159  -2.3114   True
  ExpPow    twoDt  86.9739  0.001  82.4717  91.4762   True
  LogSec Logistic   3.2064 0.2926  -1.2958   7.7087  False
  LogSec  Weibull -88.1764  0.001 -92.6786 -83.6741   True
  LogSec    twoDt   5.6112 0.0062    1.109  10.1135   True
Logistic  Weibull -91.3828  0.001  -95.885 -86.8805   True
Logistic    twoDt   2.4048 0.5758  -2.0974    6.907  False
 Weibull    twoDt  93.7876  0.001  89.2853  98.2898   True
----------------------------------------------------------
=== Fra_exc ===
   Multiple Comparison of Means - Tukey HSD, FWER=0.05
=========================================================
 group1   group2  meandiff p-adj  lower    upper   reject
---------------------------------------------------------
  ExpPow   LogSec  10.0143 0.001   4.4796  15.5491   True
  ExpPow Logistic  31.4735 0.001  25.9388  37.0083   True
  ExpPow  Weibull -54.9356 0.001 -60.4704 -49.4009   True
  ExpPow    twoDt -14.3062 0.001 -19.8409  -8.7714   True
  LogSec Logistic  21.4592 0.001  15.9245   26.994   True
  LogSec  Weibull -64.9499 0.001 -70.4847 -59.4152   True
  LogSec    twoDt -24.3205 0.001 -29.8552 -18.7857   True
Logistic  Weibull -86.4092 0.001 -91.9439 -80.8744   True
Logistic    twoDt -45.7797 0.001 -51.3144 -40.2449   True
 Weibull    twoDt  40.6295 0.001  35.0947  46.1642   True
---------------------------------------------------------
=== Pic_abi ===
```

```
C:\Users\zani\Anaconda3\envs\LPJGM_uncertainty\lib\site-
packages\scipy\stats\stats.py:3621: F_onewayBadInputSizesWarning: at least one
input has length 0
  warnings.warn(F_onewayBadInputSizesWarning('at least one input '
```

```
    Multiple Comparison of Means - Tukey HSD, FWER=0.05
===========================================================
 group1   group2  meandiff p-adj    lower    upper   reject
-----------------------------------------------------------
  ExpPow   LogSec  78.5824  0.001   72.1463  85.0186   True
  ExpPow Logistic  72.7273  0.001   66.2911  79.1634   True
  ExpPow  Weibull  -20.339  0.001  -26.7751 -13.9029   True
  ExpPow    twoDt  60.7088  0.001   54.2727  67.1449   True
  LogSec Logistic  -5.8552 0.0945  -12.2913    0.581  False
  LogSec  Weibull -98.9214  0.001 -105.3575 -92.4853   True
  LogSec    twoDt -17.8737  0.001  -24.3098 -11.4375   True
Logistic  Weibull -93.0663  0.001  -99.5024 -86.6301   True
Logistic    twoDt -12.0185  0.001  -18.4546  -5.5824   True
 Weibull    twoDt  81.0478  0.001   74.6116  87.4839   True
-----------------------------------------------------------
=== Pic_sit ===
    Multiple Comparison of Means - Tukey HSD, FWER=0.05
===========================================================
 group1   group2  meandiff p-adj    lower    upper   reject
-----------------------------------------------------------
  ExpPow   LogSec  32.6656  0.001   26.2295  39.1018   True
  ExpPow Logistic  20.0308  0.001   13.5947  26.4669   True
  ExpPow  Weibull -66.2558  0.001  -72.6919 -59.8197   True
  ExpPow    twoDt  37.2881  0.001    30.852  43.7243   True
  LogSec Logistic -12.6348  0.001  -19.0709  -6.1987   True
  LogSec  Weibull -98.9214  0.001 -105.3575 -92.4853   True
  LogSec    twoDt   4.6225 0.2857   -1.8136  11.0586  False
Logistic  Weibull -86.2866  0.001  -92.7227 -79.8505   True
Logistic    twoDt  17.2573  0.001   10.8212  23.6934   True
 Weibull    twoDt 103.5439  0.001   97.1078   109.98   True
-----------------------------------------------------------
=== Pin_hal ===
   Multiple Comparison of Means - Tukey HSD, FWER=0.05
==========================================================
 group1   group2  meandiff p-adj   lower    upper   reject
----------------------------------------------------------
  ExpPow   LogSec  56.5031  0.001  53.3168  59.6894   True
  ExpPow Logistic  82.6108  0.001  79.4245  85.7971   True
  ExpPow  Weibull  -1.9057 0.4776   -5.092   1.2806  False
  ExpPow    twoDt  29.1567  0.001  25.9704   32.343   True
  LogSec Logistic  26.1077  0.001  22.9214   29.294   True
  LogSec  Weibull -58.4088  0.001 -61.5951 -55.2225   True
  LogSec    twoDt -27.3464  0.001 -30.5327 -24.1601   True
Logistic  Weibull -84.5164  0.001 -87.7027 -81.3301   True
Logistic    twoDt  -53.454  0.001 -56.6403 -50.2677   True
 Weibull    twoDt  31.0624  0.001  27.8761  34.2487   True
----------------------------------------------------------
=== Pin_syl ===
    Multiple Comparison of Means - Tukey HSD, FWER=0.05
============================================================
 group1   group2   meandiff p-adj   lower     upper   reject
-------------------------------------------------------------
  ExpPow   LogSec  107.0033 0.001   103.817  110.1896   True
  ExpPow Logistic   41.7342 0.001   38.5479   44.9205   True
  ExpPow  Weibull   -28.585 0.001  -31.7713  -25.3987   True
  ExpPow    twoDt   -0.2859   0.9   -3.4721    2.9004  False
  LogSec Logistic  -65.2692 0.001  -68.4555  -62.0829   True
  LogSec  Weibull -135.5884 0.001 -138.7747 -132.4021   True
  LogSec    twoDt -107.2892 0.001 -110.4755 -104.1029   True
Logistic  Weibull  -70.3192 0.001  -73.5055  -67.1329   True
Logistic    twoDt    -42.02 0.001  -45.2063  -38.8337   True
 Weibull    twoDt   28.2992 0.001   25.1129   31.4855   True
-------------------------------------------------------------
=== Que_coc ===
     Multiple Comparison of Means - Tukey HSD, FWER=0.05
==============================================================
 group1   group2   meandiff p-adj    lower     upper   reject
--------------------------------------------------------------
  ExpPow   LogSec  107.8849  0.001  103.9256  111.8441   True
  ExpPow Logistic  104.8811  0.001  100.9219  108.8403   True
  ExpPow  Weibull    0.5006    0.9   -3.4586    4.4599  False
  ExpPow    twoDt  120.6508  0.001  116.6916    124.61   True
  LogSec Logistic   -3.0038 0.2326    -6.963    0.9555  False
  LogSec  Weibull -107.3842  0.001 -111.3435  -103.425   True
  LogSec    twoDt    12.766  0.001    8.8067   16.7252   True
Logistic  Weibull -104.3805  0.001 -108.3397 -100.4212   True
Logistic    twoDt   15.7697  0.001   11.8105   19.7289   True
 Weibull    twoDt  120.1502  0.001   116.191  124.1094   True
--------------------------------------------------------------
=== Que_ile ===
    Multiple Comparison of Means - Tukey HSD, FWER=0.05
============================================================
 group1   group2   meandiff p-adj   lower    upper   reject
------------------------------------------------------------
  ExpPow   LogSec   97.1214 0.001   93.1622 101.0806   True
  ExpPow Logistic   88.8611 0.001   84.9018  92.8203   True
  ExpPow  Weibull   -6.0075 0.001   -9.9667  -2.0483   True
  ExpPow    twoDt  155.4443 0.001  151.4851 159.4035   True
  LogSec Logistic   -8.2603 0.001  -12.2196  -4.3011   True
  LogSec  Weibull -103.1289 0.001 -107.0881 -99.1697   True
  LogSec    twoDt   58.3229 0.001   54.3637  62.2821   True
Logistic  Weibull  -94.8686 0.001  -98.8278 -90.9094   True
Logistic    twoDt   66.5832 0.001    62.624  70.5425   True
 Weibull    twoDt  161.4518 0.001  157.4926  165.411   True
------------------------------------------------------------
=== Que_pub ===
    Multiple Comparison of Means - Tukey HSD, FWER=0.05
=============================================================
 group1   group2   meandiff p-adj    lower    upper   reject
------------------------------------------------------------
  ExpPow   LogSec   96.1202  0.001   92.1609 100.0794   True
  ExpPow Logistic   75.8448  0.001   71.8856   79.804   True
  ExpPow  Weibull   -6.5081  0.001  -10.4674  -2.5489   True
  ExpPow    twoDt   72.0901  0.001   68.1309  76.0493   True
  LogSec Logistic  -20.2753  0.001  -24.2346 -16.3161   True
  LogSec  Weibull -102.6283  0.001 -106.5875 -98.6691   True
  LogSec    twoDt    -24.03  0.001  -27.9893 -20.0708   True
Logistic  Weibull  -82.3529  0.001  -86.3122 -78.3937   True
Logistic    twoDt   -3.7547 0.0728   -7.7139   0.2045  False
 Weibull    twoDt   78.5982  0.001    74.639  82.5575   True
------------------------------------------------------------
=== Que_rob ===
    Multiple Comparison of Means - Tukey HSD, FWER=0.05
============================================================
 group1   group2   meandiff p-adj   lower     upper   reject
------------------------------------------------------------
  ExpPow   LogSec  104.3805 0.001  100.4212  108.3397   True
  ExpPow Logistic  111.3892 0.001    107.43  115.3485   True
  ExpPow  Weibull       0.0   0.9   -3.9592    3.9592  False
  ExpPow    twoDt   54.8185 0.001   50.8593   58.7778   True
  LogSec Logistic    7.0088 0.001    3.0495    10.968   True
  LogSec  Weibull -104.3805 0.001 -108.3397 -100.4212   True
  LogSec    twoDt   -49.562 0.001  -53.5212  -45.6027   True
Logistic  Weibull -111.3892 0.001 -115.3485   -107.43   True
Logistic    twoDt  -56.5707 0.001  -60.5299  -52.6115   True
 Weibull    twoDt   54.8185 0.001   50.8593   58.7778   True
------------------------------------------------------------
=== Til_cor ===
    Multiple Comparison of Means - Tukey HSD, FWER=0.05
===========================================================
 group1   group2  meandiff p-adj   lower     upper   reject
-----------------------------------------------------------
  ExpPow   LogSec  21.8798 0.001   15.4437   28.3159   True
  ExpPow Logistic 123.2666 0.001  116.8304  129.7027   True
  ExpPow  Weibull -69.3374 0.001  -75.7736  -62.9013   True
  ExpPow    twoDt  52.3883 0.001   45.9522   58.8244   True
  LogSec Logistic 101.3867 0.001   94.9506  107.8229   True
  LogSec  Weibull -91.2173 0.001  -97.6534  -84.7811   True
  LogSec    twoDt  30.5085 0.001   24.0723   36.9446   True
Logistic  Weibull -192.604 0.001 -199.0401 -186.1679   True
Logistic    twoDt -70.8783 0.001  -77.3144  -64.4421   True
 Weibull    twoDt 121.7257 0.001  115.2896  128.1619   True
-----------------------------------------------------------
=== Ulm_gla ===
   Multiple Comparison of Means - Tukey HSD, FWER=0.05
=========================================================
 group1   group2  meandiff p-adj  lower    upper   reject
---------------------------------------------------------
  ExpPow   LogSec  66.1072 0.001  63.0512  69.1631   True
  ExpPow Logistic  51.7753 0.001  48.7194  54.8313   True
  ExpPow  Weibull -15.4939 0.001 -18.5498 -12.4379   True
  ExpPow    twoDt  89.6062 0.001  86.5502  92.6622   True
  LogSec Logistic -14.3318 0.001 -17.3878 -11.2758   True
  LogSec  Weibull  -81.601 0.001  -84.657 -78.5451   True
  LogSec    twoDt   23.499 0.001   20.443   26.555   True
Logistic  Weibull -67.2692 0.001 -70.3252 -64.2132   True
Logistic    twoDt  37.8309 0.001  34.7749  40.8868   True
 Weibull    twoDt 105.1001 0.001 102.0441  108.156   True
---------------------------------------------------------
```

[25]:

```
print('Default NegExp kernel \n--> minimum RMSE: '+str(round(min(best_EVA_df.
 ↪rmse),2))+'% for '+
      str(best_EVA_df.loc[best_EVA_df.rmse == min(best_EVA_df.rmse)].Species.
 ↪tolist())+
     '\n--> maximum RMSE: '+str(round(max(best_EVA_df.rmse),2))+'% for '+
      str(best_EVA_df.loc[best_EVA_df.rmse == max(best_EVA_df.rmse)].Species.
 ↪tolist()))
print('Fat-tailed kernels \n--> minimum RMSE: '+str(round(min(rmse_all_df.
 ↪rmse),2))+'% with '+
      str(rmse_all_df.loc[rmse_all_df.rmse == min(rmse_all_df.rmse)].kernel.
 ↪tolist())+' for '+
      str(rmse_all_df.loc[rmse_all_df.rmse == min(rmse_all_df.rmse)].Species.
 ↪tolist())+
     '\n--> maximum RMSE: '+str(round(max(rmse_all_df.rmse),2))+'% with '+
      str(rmse_all_df.loc[rmse_all_df.rmse == max(rmse_all_df.rmse)].kernel.
 ↪tolist())+' for '+
      str(rmse_all_df.loc[rmse_all_df.rmse == max(rmse_all_df.rmse)].Species.
 ↪tolist()))
```

```
Default NegExp kernel
--> minimum RMSE: 2.66% for ['Abi_alb']
--> maximum RMSE: 191.48% for ['Pin_hal']
Fat-tailed kernels
--> minimum RMSE: 7.99% with ['twoDt'] for ['Abi_alb']
--> maximum RMSE: 34.32% with ['Logistic'] for ['Pin_syl']
```

9.8 Best kernel shapes per species

[26]:

```
## Create dataframe with:
# simulated and observed (75th percentile) migration rates
# migration parameters
# life-history traits (dispersal syndromes and kernel shape categories)
# Add species-specific migration parameters
best_all_df = rmse_all_df
best_all_df['FEC_max'] = KA_input.groupby('Species').mean().FEC_max.tolist()
best_all_df['GERM_p'] = KA_input.groupby('Species').mean().GERM_p.tolist()
best_all_df['SDD_d'] = KA_input.groupby('Species').mean().SDD_d.tolist()
best_all_df['LDD_d'] = KA_input.groupby('Species').mean().LDD_d.tolist()

# Add best default kernel (from EVA)
EVA_opt_df = best_EVA_df.loc[best_EVA_df['Parameter_combo'] == 'all_MAX_opt']
species_best_EVA = EVA_opt_df.Species.to_list()
for sp in species_best_EVA :
    best_all_df.loc[rmse_all_df['Species'] == sp, 'kernel'] = 'NegExp'
    best_all_df.loc[rmse_all_df['Species'] == sp, 'shape_par'] = 'NA'
    best_all_df.loc[rmse_all_df['Species'] == sp, 'sim'] = float(EVA_opt_df.
 ↪loc[EVA_opt_df['Species'] == sp].sim)
    best_all_df.loc[rmse_all_df['Species'] == sp, 'rmse'] = float(EVA_opt_df.
 ↪loc[EVA_opt_df['Species'] == sp].rmse)
    best_all_df.loc[rmse_all_df['Species'] == sp, 'LDD_d'] = float(EVA_opt_df.
 ↪loc[EVA_opt_df['Species'] == sp].LDD_d)

# Add species-specific dispersal syndromes (cf. Table 1)
best_all_df['dispersal_syndrome'] = ['Mainly-wind','Wind/Animal + LDD','Wind/
 ↪Animal + LDD','Wind/Animal + LDD',
                                        'Wind/Animal + LDD','Hoarding␣
 ↪animals','Wind/Animal + LDD','Wind/Animal + LDD',
                                        'Mainly-wind','Wind/Animal +␣
 ↪LDD','Mainly-wind','Hoarding animals',
                                        'Hoarding animals','Hoarding␣
 ↪animals','Hoarding animals','Hoarding animals',
                                        'Mainly-wind']

# Add simplified kernel shape description
kernel_functions = rmse_all_df.kernel.tolist()
kernel_shape = ['Fat-tailed (concave)' if (kernel_fun == 'twoDt' or kernel_fun␣
 ↪== 'ExpPow')
                else ('Exponentially-bounded' if (kernel_fun == 'NegExp')
                else 'Fat-tailed (convex)') for kernel_fun in kernel_functions]
best_all_df['kernel_shape'] = kernel_shape

# Export to CSV
best_all_df.to_csv('Table_B2.csv', sep=';', index=False)
best_all_df
```

```
[26]:
    Species    kernel shape_par       rmse    sim  obs_75ile  FEC_max  \
0   Abi_alb    NegExp        NA   2.663086  305.0     286.75     50.0
1   Bet_pen    LogSec      0.29  19.705736  778.0     734.25  11775.0
```

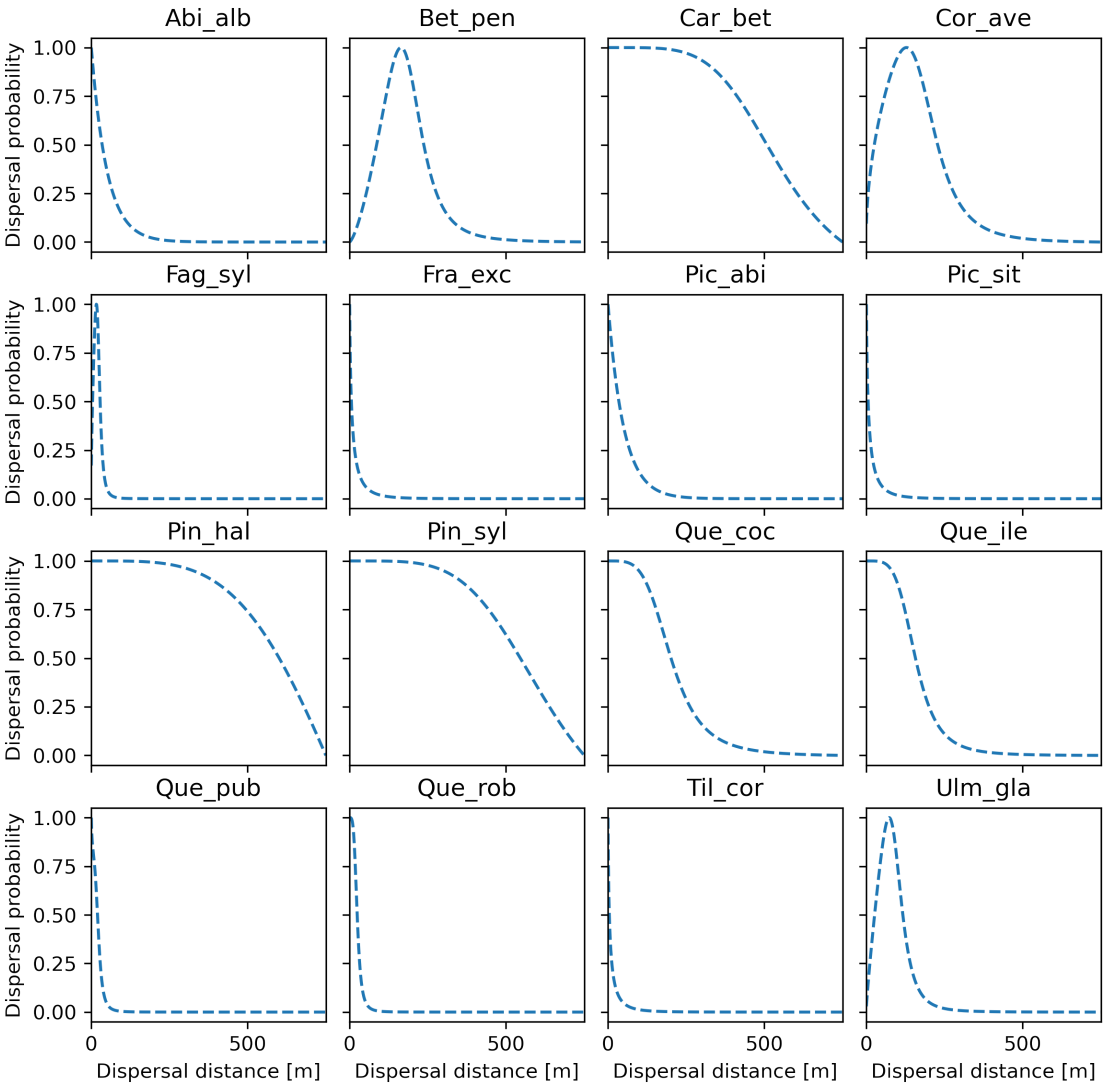
```
2   Bet_pub     LogSec      0.29  19.705736   778.0     734.25  11775.0
3   Car_bet   Logistic      4.75  19.629633   778.0     874.25    705.0
4   Cor_ave     LogSec       0.4  12.212030  1200.0    1374.25      6.0
5   Fag_syl     LogSec     0.375  14.190301   270.0     274.25     29.0
6   Fra_exc     ExpPow       0.5  25.751338   374.0     424.25     42.0
7   Pic_abi     NegExp        NA   9.206669   352.0     411.75    163.0
8   Pic_sit     ExpPow       0.5  31.145396   327.0     411.75     50.0
9   Pin_hal   Logistic       4.0  29.198774   887.0    1274.25     22.0
10  Pin_syl   Logistic       4.5  34.321916   800.0    1274.25     43.0
11  Que_coc   Logistic      4.15  21.114749   461.0     449.25      5.0
12  Que_ile   Logistic       4.5  15.253691   419.0     449.25     50.0
13  Que_pub     LogSec     0.525  14.688697   410.0     449.25     50.0
14  Que_rob     LogSec       0.5  19.533229   452.0     449.25     50.0
15  Til_cor     ExpPow      0.45  31.412574   311.0     411.75    720.0
16  Ulm_gla     LogSec      0.35  17.118000   748.0     886.75    725.0

    GERM_p  SDD_d   LDD_d dispersal_syndrome           kernel_shape
0     46.0  100.0   710.0        Mainly-wind  Exponentially-bounded
1     19.0  200.0   475.0  Wind/Animal + LDD    Fat-tailed (convex)
2     19.0  200.0   475.0  Wind/Animal + LDD    Fat-tailed (convex)
3     80.0  100.0   425.0  Wind/Animal + LDD    Fat-tailed (convex)
4     60.0  200.0  1500.0  Wind/Animal + LDD    Fat-tailed (convex)
5     71.0   25.0   200.0   Hoarding animals    Fat-tailed (convex)
6     60.0  100.0   725.0  Wind/Animal + LDD   Fat-tailed (concave)
7     80.0  100.0   780.0  Wind/Animal + LDD  Exponentially-bounded
8     75.0  100.0   800.0        Mainly-wind   Fat-tailed (concave)
9     60.0  100.0   250.0  Wind/Animal + LDD    Fat-tailed (convex)
10    91.0  100.0   250.0        Mainly-wind    Fat-tailed (convex)
11    70.0   25.0   200.0   Hoarding animals    Fat-tailed (convex)
12    90.0   25.0   200.0   Hoarding animals    Fat-tailed (convex)
13    90.0   25.0   200.0   Hoarding animals    Fat-tailed (convex)
14    95.0   25.0   200.0   Hoarding animals    Fat-tailed (convex)
15    55.0  100.0   374.0   Hoarding animals   Fat-tailed (concave)
16    65.0  100.0   350.0        Mainly-wind    Fat-tailed (convex)
```

[27]:

```
# Define kernel functions
def neg_exponential(dist, sdd, ldd) :
    a1 = sdd / 2
    a2 = ldd / 2
    if (a1 <= 0):
        return np.zeros(len(dist))
    dist_prob_sdd = 1 / (2 * np.pi * a1**2) * np.exp(-dist / a1)
    dist_prob_ldd = 1 / (2 * np.pi * a2**2) * np.exp(-dist / a2)
    dist_prob = short_range_disp_frac*dist_prob_sdd + (1 -␣
 ↪short_range_disp_frac)*dist_prob_ldd
    return scaler_0_1(dist_prob)
```

```
def exp_power(dist, sdd, ldd, b) :
    a1 = (gamma(2/b) / gamma(3/b)) * sdd
    a2 = (gamma(2/b) / gamma(3/b)) * ldd
    if (a1 == 0. or b <= 0.):
        return np.zeros(len(dist))
    dist_prob_sdd = b / (2 * np.pi * a1**2 * gamma(2/b)) * np.exp(-(dist**b /␣
 ↪a1**b))
    dist_prob_ldd = b / (2 * np.pi * a2**2 * gamma(2/b)) * np.exp(-(dist**b /␣
 ↪a2**b))
    dist_prob = short_range_disp_frac*dist_prob_sdd + (1 -␣
 ↪short_range_disp_frac)*dist_prob_ldd
    return scaler_0_1(dist_prob)

def logistic(dist, sdd, ldd, b) :
    a1 = (gamma(2/b) * gamma(1 - 2/b) / gamma(3/b) * gamma(1 - 3/b)) * sdd
    a2 = (gamma(2/b) * gamma(1 - 2/b) / gamma(3/b) * gamma(1 - 3/b)) * ldd
    if (a1 <= 0. or b <= 2.) :
        return np.zeros(len(dist))
    dist_prob_sdd = b / (2 * np.pi * a1**2 * gamma(2/b) * gamma(1 - 2/b)) * (1 +␣
 ↪dist**b / a1**b)**-1
    dist_prob_ldd = b / (2 * np.pi * a2**2 * gamma(2/b) * gamma(1 - 2/b)) * (1 +␣
 ↪dist**b / a2**b)**-1
    dist_prob = short_range_disp_frac*dist_prob_sdd + (1 -␣
 ↪short_range_disp_frac)*dist_prob_ldd
    return scaler_0_1(dist_prob)

def log_sech(dist, sdd, ldd, b) :
    a1 = sdd
    a2 = ldd
    if (a1 <= 0. or b <= 0.) :
        return np.zeros(len(dist))
    dist_prob_sdd = 1 / (np.pi**2 * b * dist**2) / ((dist / a1)**(1 / b) + (dist␣
 ↪/ a1)**(-1 / b))
    dist_prob_ldd = 1 / (np.pi**2 * b * dist**2) / ((dist / a2)**(1 / b) + (dist␣
 ↪/ a2)**(-1 / b))
    dist_prob = short_range_disp_frac*dist_prob_sdd + (1 -␣
 ↪short_range_disp_frac)*dist_prob_ldd
    return scaler_0_1(dist_prob)

f, axes = plt.subplots(4, 4, sharex=True, sharey=True)
(ax1, ax2, ax3, ax4), (ax5, ax6, ax7, ax8), (ax9, ax10, ax11, ax12), (ax13,␣
 ↪ax14, ax15, ax16) = axes
f.set_figheight(7.25*1.25)
f.set_figwidth(7.25*1.25)
f.dpi = 300
```

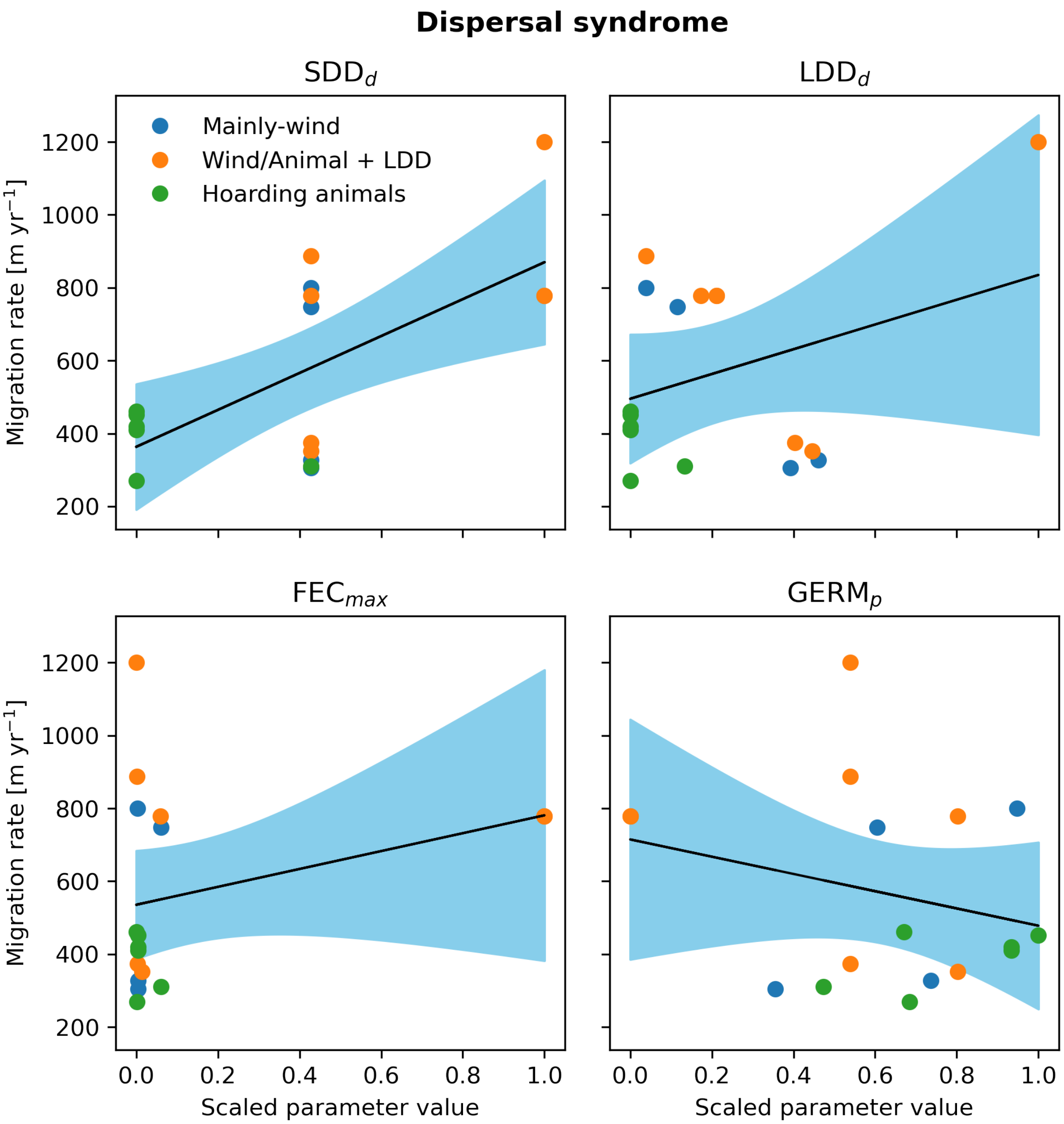
```
# Drop one of the Betula spp. as they are identical
bestkernels_sp_df = best_all_df.copy()
bestkernels_sp_df.drop(bestkernels_sp_df[bestkernels_sp_df.Species == 'Bet_pub'].
 ↪index, inplace=True)

species = bestkernels_sp_df.Species
distance_range = np.linspace(1.,750.,750)
short_range_disp_frac = 0.99

for sp,ax in zip(species,axes.flatten()):
    df_sub = bestkernels_sp_df[bestkernels_sp_df.Species == sp]

    if df_sub.kernel.item() == 'NegExp':
        x = neg_exponential(distance_range, df_sub.SDD_d.item(), df_sub.LDD_d.
 ↪item())
    if df_sub.kernel.item() == 'ExpPow':
        x = exp_power(distance_range, df_sub.SDD_d.item(), df_sub.LDD_d.item(),␣
 ↪df_sub.shape_par.item())
    if df_sub.kernel.item() == 'Logistic':
        x = logistic(distance_range, df_sub.SDD_d.item(), df_sub.LDD_d.item(),␣
 ↪df_sub.shape_par.item())
    if df_sub.kernel.item() == 'LogSec':
        x = log_sech(distance_range, df_sub.SDD_d.item(), df_sub.LDD_d.item(),␣
 ↪df_sub.shape_par.item())

    #ax.plot(distance_range,x,color='black',ls='--')
    ax.plot(distance_range,x,ls='--')
    ax.grid(False)
    ax.set_xlim(-0.05,distance_range.max())
    ax.set_ylim(-0.05,1.05)
    ax.set_title(sp)

ax13.set_xlabel('Dispersal distance [m]')
ax14.set_xlabel('Dispersal distance [m]')
ax15.set_xlabel('Dispersal distance [m]')
ax16.set_xlabel('Dispersal distance [m]')
ax1.set_ylabel('Dispersal probability')
ax5.set_ylabel('Dispersal probability')
ax9.set_ylabel('Dispersal probability')
ax13.set_ylabel('Dispersal probability')
f.subplots_adjust(wspace=0.1)

plt.savefig('Figure_6.png', bbox_inches='tight', dpi=300)
```



9.9 Migration rate and life-history traits

[28]:

```
# Function to plot migration rate vs. parameter values
# Color points by group
def scatterplot_group(df, group_name='dispersal_syndrome'):
    f, axes = plt.subplots(2, 2, sharex=True, sharey=True)
    (ax1, ax2), (ax3, ax4) = axes
    f.set_figheight(7.25)
    f.set_figwidth(7.25)
    f.dpi = 300

    headers = ['SDD_d','LDD_d','FEC_max','GERM_p']
    for header,ax in zip(headers,axes.flatten()):
        # Select y, x and scale x
        y = df.sim
        x = df[header]
        x = scaler_0_1(x)
        df[header] = x

        # Select group-color
        group_lab = df[group_name].unique()
        for g_lab in group_lab:
            group_df = df[df[group_name] == g_lab]
            ys = group_df.sim
            xs = group_df[header]
            ax.plot(xs, ys, marker="o", linestyle="", label=g_lab)

        # Calculate statistics
        # ref: https://stackoverflow.com/questions/27164114/
↪show-confidence-limits-and-prediction-limits-in-scatter-plot
        slope, intercept = np.polyfit(x, y, 1)
        y_model = np.polyval([slope, intercept], x)
        x_mean = np.mean(x)
        y_mean = np.mean(y)
        n = x.size #number of samples
        m = 2 #number of parameters
        dof = n - m  #degrees of freedom
        t = stats.t.ppf(0.975, dof) #Students statistic of interval confidence
        residual = y - y_model
        std_error = (np.sum(residual**2) / dof)**.5 #standard deviation of the␣
↪error
        x_line = np.linspace(np.min(x), np.max(x), 100)
        y_line = np.polyval([slope, intercept], x_line)
        ci = t * std_error * (1/n + (x_line - x_mean)**2 / np.sum((x -␣
↪x_mean)**2))**.5

        # Draw regression line and confidence interval
        ax.plot(x, slope*x+intercept, color='black', linewidth=1)
        ax.fill_between(x_line, y_line + ci, y_line - ci, color = 'skyblue')

        # Add headers
        if header == 'SDD_d' :
            ax.set_title('SDD$_d$')
            ax.legend(frameon=False)
        if header == 'LDD_d' :
            ax.set_title('LDD$_d$')
        if header == 'FEC_max' :
            ax.set_title('FEC$_{max}$')
        if header == 'GERM_p' :
            ax.set_title('GERM$_p$')

    ax1.set_ylabel('Migration rate [m yr$^{-1}$]')
    ax3.set_ylabel('Migration rate [m yr$^{-1}$]')
    ax3.set_xlabel('Scaled parameter value')
    ax4.set_xlabel('Scaled parameter value')

    # Add general title
    if (group_name == 'dispersal_syndrome'):
        f.suptitle('Dispersal syndrome', fontweight='bold')
    if (group_name == 'kernel_shape'):
        f.suptitle('Kernel shape', fontweight='bold')
    f.subplots_adjust(wspace=0.1,top=0.91)

# Groups: dispersal_syndrome, kernel_shape
scatterplot_group(df=best_all_df, group_name='dispersal_syndrome')
plt.savefig('Figure_S4.png', bbox_inches='tight', dpi=300)
```



10 Uncertainty analysis

In order to select the best model structure, we quantified the utility U of each model framework as a synthesis of both sensitivity and error (Snowling and Kramer, 2001):

$$U = \sqrt{2} - \sqrt{(\frac{S}{S_{max}})^2 + (\frac{E}{E_{max}})^2}$$

where the sensitivity S and error E are calculated according to the equations above for II_2 and $RMSE$ across species and parameters, respectively, whereas S_{max} and E_{max} are the maximum sensitivity and error across models, respectively.

Additionally, we calculated an index of model complexity:

$$C = \sum_{j=1}^{N} \sum_{l=1}^{n_j} p_l r_l$$

where N is the number of state variable, n_j is the number of processes implemented for each state variable j, p_l is the number of parameters of each process l, and r_l is the number of equations used to formulate each process.

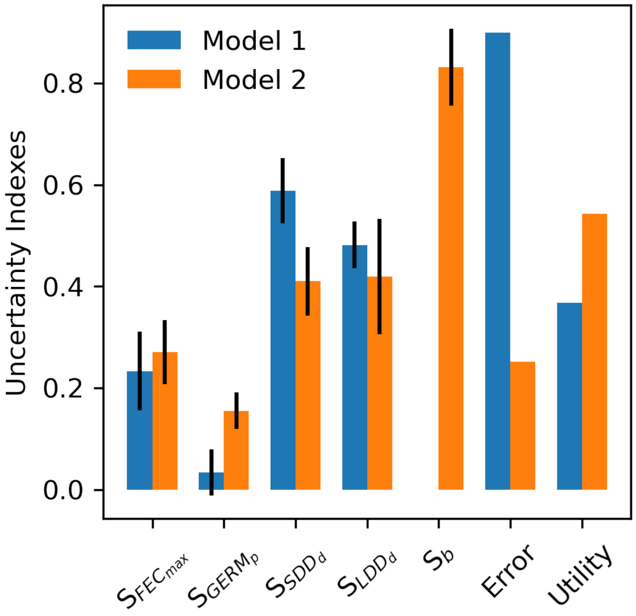
[29]:

```
## Calculate Sensitivities across species for both models
S_model1 = SA_input.groupby('Parameters').agg(['mean',f_ste]).T.loc['II2']
SA_fatkernels_input['II2'] = (SA_fatkernels_input.MigRate_Max -␣
 ↪SA_fatkernels_input.MigRate_Min) / SA_fatkernels_input.MigRate_Max
S_model2 = SA_fatkernels_input.groupby('Parameters').agg(['mean',f_ste]).T.
 ↪loc['II2']

# Create dataframe to plot
S_toPlot = pd.concat([S_model1, S_model2])
S_toPlot.index = ['mean_model1','ste_model1','mean_model2','ste_model2']
S_toPlot.columns = ['S_FEC_max','S_GERM_p','S_LDD_d','S_SDD_d']

# Calculate II2 for shape parameter (only model 2)
species = KA_input.Species.unique()
II2s = np.zeros(len(species))
pos = 0
for sp in species:
    # Subset for best kernel per species
    sp_sub = KA_input[KA_input['Species'] == sp]
    best_kernel_sp = rmse_all_df.loc[rmse_all_df['Species'] == sp, 'kernel'].
 ↪tolist()
    best_kernel_sub = sp_sub[sp_sub['kernel_fun'] == str(best_kernel_sp[0])]

    # Get the minimum and maximum simulated migration speed
    y_min = best_kernel_sub.loc[best_kernel_sub.SIM == min(best_kernel_sub.SIM)].
 ↪SIM.tolist()
    y_min = y_min[0]
    y_max = best_kernel_sub.loc[best_kernel_sub.SIM == max(best_kernel_sub.SIM)].
 ↪SIM.tolist()
    y_max = y_max[0]

    # Calculate II2 per species
    II2 = (y_max - y_min) / y_max
    II2s[pos] = II2
    pos = pos + 1
```

```
# Calculate mean and ste II2 across species (shape parameter)
II2_mean_shapepar = np.mean(II2s)
II2_ste_shapepar = II2s.std() / np.sqrt(len(II2s))
S_toPlot['S_shape_par'] = [0,0,II2_mean_shapepar,II2_ste_shapepar] #only model 2

## Calculate mean Sensitivity S across parameters
S_mean_model1 = np.mean(S_toPlot.T.mean_model1)
S_mean_model2 = np.mean(S_toPlot.T.mean_model2)

# Calculate average of observed 75thile migration rates across species
obs_mean = np.mean(best_EVA_df.obs_75ile)

## Calculate Error E across species
E_model1 = rmse_perc(best_EVA_df.sim, best_EVA_df.obs_75ile, obs_mean) / 100
E_model2 = rmse_perc(rmse_all_df.sim, rmse_all_df.obs_75ile, obs_mean) / 100
S_toPlot['Error'] = [E_model1, 0, E_model2, 0]

# Find S_max (rounded-up maximum)
S_max = max(S_mean_model1, S_mean_model2)
S_max = round(S_max,1) + 0.1

# Find E_max (rounded-up maximum)
E_max = max(E_model1, E_model2)
E_max = round(E_max,1) + 0.1

## Calculate Utility U
U_model1 = np.sqrt(2) - np.sqrt((S_mean_model1 / S_max)**2 + (E_model1 /␣
 ↪E_max)**2)
U_model2 = np.sqrt(2) - np.sqrt((S_mean_model2 / S_max)**2 + (E_model2 /␣
 ↪E_max)**2)
S_toPlot['Utility'] = [U_model1, 0, U_model2, 0]
S_toPlot = S_toPlot.T

# Plot Figure 6
labels =␣
 ↪['S$_{FEC_{max}}$','S$_{GERM_{p}}$','S$_{SDD_{d}}$','S$_{LDD_{d}}$','S$_{b}$','Error','Utilit
x = np.arange(len(labels))
width = 0.35

f, ax = plt.subplots()
f.set_figheight(3.54)
f.set_figwidth(3.54)
f.dpi = 300

rects1 = ax.bar(x - width/2, S_toPlot['mean_model1'], width,␣
 ↪yerr=S_toPlot['ste_model1'], label='Model 1')
```

```
rects2 = ax.bar(x + width/2, S_toPlot['mean_model2'], width,␣
 ↪yerr=S_toPlot['ste_model2'], label='Model 2')

ax.set_xticks(x)
ax.set_xticklabels(labels)
ax.set_ylabel('Uncertainty Indexes')
ax.legend(frameon=False)
plt.xticks(rotation=45)

# Export figure
plt.savefig('Figure_7.png', bbox_inches='tight', dpi=300)

# Calculate Complexity C
C_model1 = 5
C_model2 = 6

# Print results
print('Model 1 | Complexity: '+str(round(C_model1,2))+'; Sensitivity:␣
 ↪'+str(round(S_mean_model1,2))+
      '; Error: '+str(round(E_model1*100,2))+'%; Utility:␣
 ↪'+str(round(U_model1,2))+
     '\nModel 2 | Complexity: '+str(round(C_model2,2))+'; Sensitivity:␣
 ↪'+str(round(S_mean_model2,2))+
      '; Error: '+str(round(E_model2*100,2))+'%; Utility:␣
 ↪'+str(round(U_model2,2)))
```

```
Model 1 | Complexity: 5; Sensitivity: 0.27; Error: 89.94%; Utility: 0.37
Model 2 | Complexity: 6; Sensitivity: 0.42; Error: 25.14%; Utility: 0.54
```


Model 1
Model 2
Uncertainty Indexes
$S_{FEC_{max}}$
S_{GERM_p}
S_{SDD_d}
S_{LDD_d}
S_b
Error
Utility